

The Linux Kernel Primer

Linux内核编程

Claudia Salzberg Rodriguez

[美] Gordon Fischer

Steven Smolski

著

陈莉君 贺炎 刘霞林 译



人民邮电出版社
POSTS & TELECOM PRESS

“本书详细比较了x86和PPC体系结构下的汇编程序，并介绍了分析工具。非常不错！”

——亚马逊读者评论

The Linux Kernel Primer

Linux内核编程

Linux操作系统博大精深，有为数不少的子系统，它们之间的关系更是错综复杂而耐人寻味。即使你拥有Linux内核源代码，仍会感到些许迷茫，该从何处开始阅读？应留意哪些地方？该以怎样的顺序浏览代码？本书通过由表及里的讲解，引导你了解内核的各个部分，理清各模块互相之间怎样关联。

本书共分为三部分。第一部分讲解必要的工具和背景，便于对Linux内核展开进一步的探索；第二部分介绍了每个内核子系统所涉及的基本概念，并分析了执行子系统功能的代码；第三部分描述与内核进行交互的有效途径。

本书适合各个层次的系统程序员、Linux应用程序开发人员阅读。

- 源代码级内核分析
- 自上而下纵观Linux内核的基本框架
- 示例丰富、注释详尽



图灵网站: www.turingbook.com 热线: (010)51095186转604

反馈/投稿/推荐信箱: contact@turingbook.com

有奖勘误: debug@turingbook.com

分类建议 计算机/操作系统/Linux

人民邮电出版社网址: www.ptpress.com.cn



ISBN 978-7-115-25194-7



9 787115 251947 >

ISBN 978-7-115-25194-7

定价: 75.00元

TURING

Linux/Unix系列

The Linux Kernel Primer

Linux内核编程

Claudia Salzberg Rodriguez

[美] Gordon Fischer

著

Steven Smolski

陈莉君 贺炎 刘霞林 译



人民邮电出版社

北京

图书在版编目 (CIP) 数据

Linux内核编程 / (美) 罗德里格斯
(Rodriguez, C. S.), (美) 费舍尔 (Fischer, G.), (美)
斯莫斯基 (Smolski, S.) 著; 陈莉君, 贺炎, 刘霞林译.
— 北京: 人民邮电出版社, 2011. 6
(图灵程序设计丛书)
书名原文: The Linux Kernel Primer: A Top-Down
Approach for x86 and PowerPC Architectures
ISBN 978-7-115-25194-7

I. ①L… II. ①罗… ②费… ③斯… ④陈… ⑤贺…
⑥刘… III. ①Linux操作系统—程序设计 IV.
①TP316.89

中国版本图书馆CIP数据核字(2011)第055985号

内 容 提 要

本书以 Linux 操作系统为基础, 详细介绍了 Linux 内核子系统, 并辅以大量内核源代码和示例程序进行演示, 引领读者深入 Linux 内核。本书的主要内容包括: Linux 基础知识、内核探索工具集、进程的整个生命周期、内存区、页面、Slab 分配器、用于输入/输出的各种设备、文件系统、抢占、自旋锁、信号量、内核引导、构建 Linux 内核, 以及向内核添加代码等, 同时还简单介绍了一些应用工具和实用程序。每章末尾都给出一些练习, 涉及内核运行的操作及工作原理。

本书适合对 Linux 内核感兴趣的各层次读者, 无论对 Linux 初学者还是 Linux 程序开发人员, 本书都是一本很有价值的参考书。

图灵程序设计丛书

Linux内核编程

◆ 著 [美] Claudia Salzberg Rodriguez Gordon Fischer
Steven Smolski

译 陈莉君 贺 炎 刘霞林

责任编辑 王军花

执行编辑 李 静

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
中国铁道出版社印刷厂印刷

◆ 开本: 800×1000 1/16

印张: 26

字数: 614千字

2011年6月第1版

印数: 1-3 000册

2011年6月北京第1次印刷

著作权合同登记号 图字: 01-2010-4025号

ISBN 978-7-115-25194-7

定价: 75.00元

读者服务热线: (010)51095186转604 印装质量热线: (010)67129223

反盗版热线: (010)67171154

版 权 声 明

Authorized translation from the English language edition, entitled *The Linux Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures* by Claudia Salzberg Rodriguez, Gordon Fischer, Steven Smolski, published by Pearson Education, Inc., publishing as Prentice Hall, Copyright ©2006 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. and POSTS & TELECOM PRESS Copyright © 2011.

本书中文简体字版由 Pearson Education Asia Ltd.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。
版权所有，侵权必究。



译者序

溯本求源，从本书开始！

打开 Linux 内核源代码，我们可以看到熟悉的 C 语言函数和一些陌生的汇编代码。应该说，Linux 内核入门是不容易的，原因在于其代码规模庞大，而且涉及的层面众多。规模一大，就不易现出本来面目，枝叶庞杂，自然不容易找到着手之处；层面一多，就会让人眼花缭乱，盘根错节，怎能让人提纲挈领？

就我们的经验，本书是内核初学者（不是编程初学者）登堂入室的首选。本书三位作者有多年的行业经验：Claudia Salzberg Rodriguez 就职于 IBM Linux 技术中心，从事内核及相关编程工具的开发工作；Gordon Fischer 为很多设备开发了 Linux 和 UNIX 设备驱动程序；Steven Smolski 在半导体行业已经浸染了 26 年，开发过各种驱动程序和嵌入式系统。他们合作奉献给大家的这本内核入门书，是对 Linux 内核编程的有效指导。作者独特的由表及里的讲解方法使得内核编程更易于理解：从用户空间到内核，把内核内在的实现原理与用户级编程的基本原则相联系，系统地追踪了实现功能。这种途径有助于扩大你所了解的 Linux 知识，加深对内核组成及工作机理的理解。

在本书的翻译过程中，我们能感受到作者软硬件知识之全面、内容组织方式之独到，以及开发经验之丰富。在我们熟知的 x86 平台之外，作者还对 PowerPC 平台进行了深入讲解，不仅让基于 PowerPC 平台的开发者找到了知音，更为 x86 的开发者打开了一扇窗。

本书第 1、4、9、10 章由陈莉君翻译，第 2、5、8 章由贺炎翻译，第 3、6、7 章由刘霞林翻译，全书由陈莉君统稿。书中不妥之处和错误在所难免，望读者指正。

序 言

“有龙出没”，中世纪地图绘制者碰到未知和危险的地方就如此标记，可能你首次敲入如下命令也有这样的感觉：

```
cd /usr/src/linux ; ls
```

“我该从何处入手？”你困惑不已。“我到底在寻找什么？所有这些是怎样放在一起并真正起作用的？”

功能全面的现代操作系统庞大而复杂，有为数不少的子系统，它们之间的交互更是错综复杂而微妙难言。不错，你的确拥有 Linux 内核源代码（稍后还会详述），但是，从何处开始阅读，应留意哪些地方，该以怎样的顺序浏览代码，这些问题的答案都不是显而易见的。

本书的作用正在于此。它一步一步地让你了解到内核的各个部分，它们如何工作，互相之间怎样关联。本书作者熟知内核，将这些知识贯穿于本书的始终。在学完本书之后，你和内核至少会成为好朋友，乃至产生深厚的情意。

Linux 内核是“自由”软件。Richard Stallman 在“自由软件的定义”一文^①中，说明了软件自由的含义。有两个级别，Freedom 0，运行软件是自由的，这是最基本的自由。紧跟其后的是 Freedom 1，探究程序运行原理也是自由的。这种自由往往被忽略，实际上，这才是最重要的，因为学习的最好方式就是观察别人如何做事。在软件世界中，那就意味着阅读别人的程序，并了解他们哪方面做得好，哪方面做得不好。至少在我看来，GNU/Linux 之所以在现代计算领域变成一股强大的力量，最根本的原因之一就是 GPL 协议的自由。你在使用 GNU/Linux 的每时每刻都会感到这种自由的可贵，别忘了经常默念一下它的好处。

本书充分利用 Freedom 1，带你深入研究 Linux 内核源代码。你会看到所有的一切，有些方面的确做得不错，还有些事也并不尽人意。但是，由于有了 Freedom 1，你会看到全貌，并从中学习到很多。

况且，这也使我与“Prentice Hall 开源软件开发系列丛书”结缘，本书是丛书的第一辑。开发这个系列的想法来自于这样一个理念：阅读代码是学习编程的最好方法。如今这个世界，人们幸福地享有丰富而自由的开源软件，这些源代码正期待着（或许热切渴望着）被大家阅读、理解和赞许。该系列丛书旨在成为你软件开发学习过程中的领路人，即通过尽可能多地展示真实的代码帮助你学好编程。

^① <http://www.gnu.org/philosophy/free-sw.html>

我真诚地希望你会喜欢本书，并从中获益多多。我也希望本书能激发你的灵感，从而在自由软件和开源世界开创你自己的事业，用这种方式参与进来，那无疑是最令人愉快的了。

祝你学习愉快！

Arnold Robbins

丛书主编

新平知覺
PDG

前言

无论是一般性技术还是计算机技术，对于试图了解它们的人们来说都具有不可思议的魔力。技术的发展使其影响力不断扩大，迫使人们对一些旧的概念重新评估。Linux 操作系统已经对产业变革和商业营销方式转变做出了巨大贡献。它采用 GNU 公共许可证并与 GNU 软件良性互动，占据了中心位置，围绕开源、自由软件和开发社区思想的各种争论都离不开它。Linux 无疑是一个极其成功的典范，展现了开源操作系统无比强大的力量，其理论的魔力令世界各地的程序员们如痴如狂。

对于大多数计算机用户来说，使用 Linux 正变得越来越方便。有了各种各样的发布版、社区的支持，以及工业后盾，Linux 的应用也找到了安全的港湾，出现在大学、行业应用以及数以千计的家庭用户中。

使用大潮促进了技术支持和新功能需求的日益增长。这样一来，愈来愈多的程序员发现自己对 Linux 内核内幕感兴趣，因为大量现有的（还在快速增长的）应用需要支持不同的体系结构和种类繁多的新设备。

内核向 Power 体系结构的成功移植，也助长了 Linux 操作系统在高端服务器和嵌入式系统中的全面繁荣。许多公司购买基于 Power PC 的系统来运行 Linux，因此越来越多的人需要知道 Linux 在该体系结构上的运行机理。

适合的读者

本书的读者包括初级和经验丰富的系统程序员、Linux 的热衷者，以及应用程序的开发者，这些开发者渴望更好地理解自己的程序到底是如何工作的。只要有 C 语言知识，熟悉基本的 Linux 用法，如果想知道 Linux 如何工作，那么你就会发现这本书提供了所需的基本知识，可以说，本书是理解 Linux 内核如何工作的初级读本。

不管你是只登录过 Linux 并编写了些小程序，还是你本身就是一个系统程序员，正想深入了解某个子系统的特性，本书都会有你所要的信息。

内容组织

本书分为三部分，每部分都提供必要的知识，让读者能顺利地钻研 Linux 内幕。

第一部分提供必要的工具和背景，便于对 Linux 内核展开进一步的探索。

第 1 章回顾了 Linux 和 UNIX 的历史，对比了很多发布版，并从用户空间的角度简述各种内

核子系统。

第2章描述 Linux 内核中常用的数据结构和语言的用法，介绍 x86 和 PowerPC 体系结构的汇编语言，并简述一些工具和实用程序，可用来获取理解内核内幕所需的信息。

第二部分介绍了在每个内核子系统中所涉及的基本概念，并分析了执行子系统功能的代码。

第3章讨论了进程模型的实现。本章解释了为何引入进程，并讨论了进程往返于用户空间和内核空间时的控制流。我们还讨论了进程在内核中是如何实现的，并描述了与进程执行相关的所有数据结构。本章还介绍了中断和异常，描述了这些硬件机制在每种体系结构中是如何发生的，它们与 Linux 内核又是如何交互的。

第4章描述了 Linux 内核如何追踪和管理用户空间进程的可用内存和内核的可用内存。本章描述了内核对内存分类的方式，以及决定分配和释放内存的方式，也详细描述了缺页机制以及它怎样在硬件上执行。

第5章描述了处理器如何与其他设备进行交互，内核又是如何响应和控制这些交互的。本章还涵盖了各种设备及其在内核中的实现。

第6章概述文件和目录如何在内核中实现。本章引入了虚拟文件系统，它是用于支持多文件系统的抽象层。本章还跟踪了文件相关操作的执行，如打开和关闭文件。

第7章描述调度程序的操作，调度程序让多个进程运行起来就像系统中只有一个进程在运行一样。本章详细描述了内核如何选择执行哪一个任务，进程切换时如何与硬件进行交互。本章还叙述了什么是内核抢占，它又是怎样执行的。最后，描述了系统时钟的工作原理，内核怎样使用它计时。

第8章描述电源开和关时都发生些什么。本章对各种处理器处理内核加载的方式进行了跟踪，包括对 BIOS、Open Firmware 和 bootloader 的描述。然后，考察了内核启动和初始化时的线性顺序，涉及了前面章节中讨论的所有子系统。

第三部分，描述如何编译内核并与内核进行交互的有效途径。

第9章涵盖了编译内核所必需的工具链和所执行的对象文件的格式。还详细描述了内核源代码编译（Kernel Source Build）系统如何运作，怎样把配置选项加入内核编译系统中。

第10章描述了 `/dev/random` 操作，这在所有的 Linux 系统中都可以看到。本章用它来跟踪设备，并从更具实战性的角度介绍前面各章描述过的概念。最后介绍了如何在内核中实现自己的设备。

我们的探索方法

本书给读者介绍了理解内核的必要概念。我们遵循自顶向下的方式来组织内容，具体体现在以下两个方面。

首先，我们把内核的机理和用户空间操作的执行关联起来，因为读者对后者较熟悉，所以我们会将二者结合起来，解释内核的工作。在可能时，我们从用户空间的例子说起，并跟踪代码的执行到内核。但有时，这种跟踪方式并不有效，因为需要先介绍子系统的数据类型和子结构，而后才能解释其工作原理。在这些情况下，我们把对内核子系统的解释和它与用户空间程序如何联

系的具体例子结合起来。有双重意图：其一，当内核一方面与用户空间打交道，另一方面与硬件打交道时，突出在内核中看到的层面；其二，通过跟踪代码和事件发生的顺序来解释子系统的工作原理。我们相信，这有助于读者将内核的工作原理与自己的认识匹配起来，也有利于读者了解一个特定的功能怎样与操作系统的其余部分产生联系。

其次，我们以自顶向下的角度，考察把数据结构视作子系统操作中心，并观察它们怎样与系统管理的执行行为相联系。我们尽力刻画子系统操作中心的结构，并像追踪子系统的操作一样持续关注这些数据结构。

约定

你会在全书中看到源代码列表。右上角存放有关源代码树根的源文件位置。代码中的行号是为了方便随后对代码进行解释。我们在解释内核子系统及其工作原理时，会不断引用源代码并给予解释。

命令行选项、函数名、函数输出以及变量名都用代码体加以区分。

引入一个新概念时就采用**黑体**。



致 谢

我们在此要感谢很多人，没有他们的帮助，本书不可能出版。

Claudia Salzberg Rodriguez：首先要说的是，有许许多多的人采用了各种各样的方式帮助我们完成了这本书，但面对有限的“致谢”篇幅，却很难将那些做出突出贡献的人一一列举出来。话虽如此，我还是要在此感谢所有对 Linux 内核有所贡献的人，正是他们出于对 Linux 的热爱，用辛勤工作和无私奉献开发出这样好的操作系统。我深深地感谢给予我指点的很多引路人，他们唤醒并培育了我探究一切的好奇心，并教会了我学习的本领。我也要感谢家人永不衰竭的爱与支持，他们的热情帮我度过了疲惫不堪的时期。最后，我衷心感谢 Jose Raul，他总是及时满足我的要求，总能找出办法重新点燃我的灵感。

Gordon Fischer：我要感谢所有的编程人员，在我还是菜鸟时他们能耐心地向我解释错综复杂的 Linux 内核。我还要感谢 Grady 和 Underworld，他们向我提供了美妙的编码音乐。

我们要感谢出色的编辑 Mark L. Taub，他知晓在本书出版过程中的每一步，应该怎样去做，并引领我们在那个方向上不断努力。还要感谢他在本书的撰写过程中，总能做到通情达理、善解人意，既严格要求，又平易近人。

我们还要感谢 Jim Markham 和 Erica Jamison。感谢 Jim Markham 是因为他早期的修改意见为全书的编写奠定了基础。感谢 Erica Jamison 是因为在手稿的最后一版编辑中给我们提供了反馈意见。

我们该感谢审稿人了，他们花大量的时间阅读本书，并给出建设性意见，使本书更加完善。感谢你们敏锐的眼光和深刻的见解，你们的建议是无价的。审稿人（按字母顺序）依次是 Alessio Gaspar、Mel Gorman、Benjamin Herrenschmidt、Ron McCarty、Chet Ramey、Eric Raymond、Arnold Robbins 和 Peter Salus。

我们要感谢 Kayla Dugger，她在整个编辑校对过程中以极大的热情鼓励着我们，还要感谢 Ginny Bess，她有着犀利的编辑眼光。这里要特别感谢参与编辑、校对、排版、市场以及印刷的幕后群体，正是他们，才使本书得以顺利完成。

目 录

第 1 章 概述	1	2.1.2 查找	21
1.1 UNIX 的历史	2	2.1.3 树	22
1.2 标准和通用接口	3	2.2 汇编	24
1.3 自由软件和开放源码	3	2.2.1 PowerPC	24
1.4 Linux 发布版概览	3	2.2.2 x86	27
1.4.1 Debian	4	2.3 汇编语言示例	29
1.4.2 Red Hat/Fedora	4	2.3.1 x86 中的汇编示例	30
1.4.3 Mandriva	4	2.3.2 PowerPC 中的汇编示例	31
1.4.4 SUSE	4	2.4 内联汇编	33
1.4.5 Gentoo	4	2.4.1 输出操作数	34
1.4.6 Yellow Dog	5	2.4.2 输入操作数	34
1.4.7 其他发布版	5	2.4.3 已修改过的寄存器 (已修改 的元素列表)	34
1.5 内核版本信息	5	2.4.4 参数的编号方式	34
1.6 基于 Power 的 Linux	5	2.4.5 约束条件	34
1.7 什么是操作系统	6	2.4.6 asm	35
1.8 内核组织	7	2.4.7 __volatile__	35
1.9 Linux 内核概述	7	2.5 特殊的 C 语言用法	38
1.9.1 用户接口	7	2.5.1 asmlinkage	38
1.9.2 用户标识符	8	2.5.2 UL	39
1.9.3 文件和文件系统	8	2.5.3 内联	39
1.9.4 进程	12	2.5.4 const 和 volatile	39
1.9.5 系统调用	15	2.6 内核探索工具一览	40
1.9.6 Linux 调度程序	15	2.6.1 objdump/readelf	40
1.9.7 Linux 设备驱动程序	15	2.6.2 hexdump	41
1.10 可移植性和体系结构的相关性	16	2.6.3 nm	41
1.11 小结	16	2.6.4 objcopy	42
1.12 习题	16	2.6.5 ar	42
第 2 章 内核探索工具集	18	2.7 内核发言: 倾听来自内核的消息	42
2.1 内核中常见的数据类型	18	2.7.1 printk()	42
2.1.1 链表	18	2.7.2 dmesg	42

2.7.3	/var/log/messages	42	3.7	等待队列	86
2.8	其他奥秘	43	3.7.1	添加到等待队列	88
2.8.1	__init	43	3.7.2	等待事件	89
2.8.2	likely()和unlikely()	43	3.7.3	唤醒进程	91
2.8.3	IS_ERR和PTR_ERR	44	3.8	异步执行流程	93
2.8.4	通告程序链	44	3.8.1	异常	93
2.9	小结	45	3.8.2	中断	95
2.9.1	项目: Hellomod	45	3.9	小结	114
2.9.2	第一步: 构造Linux模块的框架	45	3.9.1	项目: 系统变量current	114
2.9.3	第二步: 编译模块	46	3.9.2	项目源码	115
2.9.4	第三步: 运行代码	47	3.9.3	运行代码	116
2.10	习题	48	3.10	习题	116
第3章	进程: 程序执行的基本模型	49	第4章	内存管理	117
3.1	程序	51	4.1	页	119
3.2	进程描述符	52	4.2	内存管理区	121
3.2.1	与进程属性相关的字段	54	4.2.1	内存管理区描述符	122
3.2.2	与调度相关的字段	55	4.2.2	内存管理区操作辅助函数	124
3.2.3	涉及进程间相互关系的字段	58	4.3	页面	124
3.2.4	与进程信任状相关的字段	59	4.3.1	请求页面的函数	124
3.2.5	与进程权能相关的字段	60	4.3.2	释放页面的函数	126
3.2.6	与进程限制相关的字段	61	4.3.3	伙伴系统	126
3.2.7	与文件系统及地址空间相关的字段	63	4.4	Slab分配器	130
3.3	进程的创建: 系统调用fork()、vfork和clone()	64	4.4.1	缓存描述符	133
3.3.1	fork()函数	65	4.4.2	通用缓存描述符	135
3.3.2	vfork()函数	66	4.4.3	Slab描述符	136
3.3.3	clone()函数	67	4.5	Slab分配器的生命周期	138
3.3.4	do_fork()函数	68	4.5.1	与Slab分配器有关的全局变量	138
3.4	进程的生命周期	70	4.5.2	创建缓存	139
3.4.1	进程的状态	70	4.5.3	创建slab与cache_grow()	144
3.4.2	进程状态的转换	71	4.5.4	Slab的销毁: 退还内存与kmem_cache_destroy()	146
3.5	进程的终止	74	4.6	内存请求路径	147
3.5.1	sys_exit()函数	75	4.6.1	kmalloc()	147
3.5.2	do_exit()函数	75	4.6.2	kmem_cache_alloc()	148
3.5.3	通知父进程和sys_wait4()	77	4.7	Linux进程的内存结构	149
3.6	了解进程的动态: 调度程序的基本构架	80	4.7.1	mm_struct	150
3.6.1	基本结构	80	4.7.2	vm_area_struct	152
3.6.2	从等待中醒来或者激活	81	4.8	进程映像的分布及线性地址空间	153
			4.9	页表	155
			4.10	缺页	156

4.10.1 x86 缺页异常	156	6.4 页缓存	216
4.10.2 缺页处理程序	157	6.4.1 address_space 结构	217
4.10.3 PowerPC 缺页异常	164	6.4.2 buffer_head 结构	219
4.11 小结	164	6.5 VFS 的系统调用和文件系统层	221
4.12 项目: 进程内存映射	165	6.5.1 open()	221
4.13 习题	166	6.5.2 close()	227
第 5 章 输入/输出	167	6.5.3 read()	229
5.1 总线、桥、端口和接口的硬件实现	167	6.5.4 write()	244
5.2 设备	171	6.6 小结	246
5.2.1 块设备概述	172	6.7 习题	246
5.2.2 请求队列和 I/O 调度	173	第 7 章 进程调度和内核同步	247
5.2.3 示例: “通用” 块设备驱动程序	180	7.1 Linux 的调度程序	248
5.2.4 设备操作	182	7.1.1 选择下一个进程	248
5.2.5 字符设备	183	7.1.2 上下文切换	253
5.2.6 网络设备	184	7.1.3 让出 CPU	261
5.2.7 时钟设备	184	7.2 内核抢占	269
5.2.8 终端设备	184	7.2.1 显式内核抢占	269
5.2.9 直接存储器存取	184	7.2.2 隐式用户抢占	270
5.3 小结	185	7.2.3 隐式内核抢占	270
5.4 项目: 创建并口驱动程序	185	7.3 自旋锁和信号量	272
5.4.1 并口的硬件	185	7.4 系统时钟: 关于时间和定时器	274
5.4.2 运行在并口上的软件	187	7.4.1 实时时钟: 现在几点了	274
5.5 习题	192	7.4.2 读取 PPC 的实时时钟	276
第 6 章 文件系统	194	7.4.3 读取 x86 的实时时钟	278
6.1 文件系统的基本概念	194	7.5 小结	280
6.1.1 文件和文件名	194	7.6 习题	280
6.1.2 文件类型	195	第 8 章 内核引导	281
6.1.3 文件的附加属性	195	8.1 BIOS 和 Open Firmware	282
6.1.4 目录和路径名	196	8.2 引导加载程序	282
6.1.5 文件操作	197	8.2.1 GRUB	283
6.1.6 文件描述符	197	8.2.2 LILO	286
6.1.7 磁盘块、磁盘分区以及实现	197	8.2.3 PowerPC 和 Yaboot	286
6.1.8 性能	198	8.3 与体系结构相关的内存初始化	287
6.2 Linux 虚拟文件系统	198	8.3.1 PowerPC 的硬件内存管理	287
6.2.1 VFS 的数据结构	200	8.3.2 基于 Intel x86 体系结构的硬件内存管理	296
6.2.2 全局链表和局部链表的引用	211	8.3.3 PowerPC 和 x86 的代码汇集	305
6.3 与 VFS 相关的结构	212	8.4 原始的 RAM 盘	305
6.3.1 fs_struct 结构	212	8.5 开始: start_kernel()	306
6.3.2 files_struct 结构	213	8.5.1 调用 lock_kernel()	307

8.5.2 调用 page_address_ init()	309	8.5.32 调用 rest_init()	348
8.5.3 调用 printk(linux_ banner)	311	8.6 init 线程 (或进程 1)	349
8.5.4 调用 setup_arch	311	8.7 小结	353
8.5.5 调用 setup_per_cpu_ areas()	315	8.8 习题	353
8.5.6 调用 smp_prepare_boot_ cpu()	316	第 9 章 构建 Linux 内核	354
8.5.7 调用 sched_init()	317	9.1 工具链	354
8.5.8 调用 build_all_ zonelists()	319	9.1.1 编译程序	355
8.5.9 调用 page_alloc_init	319	9.1.2 交叉编译	355
8.5.10 调用 parse_args()	320	9.1.3 链接程序	356
8.5.11 调用 trap_init()	322	9.1.4 ELF 二进制目标文件	356
8.5.12 调用 rcu_init()	323	9.2 内核源代码的构建	360
8.5.13 调用 init_IRQ()	323	9.2.1 解释源代码	360
8.5.14 调用 softirq_init()	324	9.2.2 构建内核映像	364
8.5.15 调用 time_init()	325	9.3 小结	369
8.5.16 调用 console_init()	326	9.4 习题	369
8.5.17 调用 profile_init()	326	第 10 章 向内核添加代码	371
8.5.18 调用 local_irq_ enable()	327	10.1 浏览源代码	371
8.5.19 配置 initrd	327	10.1.1 熟悉文件系统	371
8.5.20 调用 mem_init()	327	10.1.2 filp 和 fops	372
8.5.21 调用 late_time_init()	333	10.1.3 用户空间和内核空间	374
8.5.22 调用 calibrate_delay()	333	10.1.4 等待队列	375
8.5.23 调用 pgtable_cache_ init()	334	10.1.5 工作队列及中断	378
8.5.24 调用 buffer_init()	335	10.1.6 系统调用	380
8.5.25 调用 security_scaffol- ding_startup()	336	10.1.7 其他类型的驱动程序	380
8.5.26 调用 vfs_caches_init()	336	10.1.8 设备模型和 sysfs 文件 系统	383
8.5.27 调用 radix_tree_ init()	343	10.2 编写代码	386
8.5.28 调用 signal_init()	344	10.2.1 设备基础	386
8.5.29 调用 page_writeback_ init()	344	10.2.2 符号输出	388
8.5.30 调用 proc_root_init()	346	10.2.3 IOCTL	388
8.5.31 调用 init_idle()	347	10.2.4 轮询与中断	391
		10.2.5 工作队列和 tasklet	395
		10.2.6 增加系统调用的代码	396
		10.3 构建和调试	398
		10.4 小结	399
		10.5 习题	400

第 1 章

概 述

本章内容

- UNIX的历史
- 标准和通用接口
- 自由软件和开放源码
- Linux发布版概览
- 内核版本信息
- 基于Power的Linux
- 什么是操作系统
- 内核组织
- Linux内核概述
- 可移植性和体系结构的相关性

Linux 操作系统诞生于 1991 年，是 Linus Torvalds 学生时代业余爱好的产物。与当下的迅猛发展势头相比，当初的 Linux 显得微不足道。Linux 刚开发时运行在具有 AT 硬盘、x86 体系结构的微处理器计算机上。第一个发布版支持 bash shell 和 gcc 编译器。那时还不关心其可移植性，也没有设想其在学术界和工业界是否能够广泛应用，更没有商业计划或远景宣言。然而，从第一天起，它就是免费可得的。

从早期的 beta 版开始，Linux 就在 Linus 的指导和维护下，变成了一个协作项目。这填补了一项空白，即黑客们需要一个运行在 x86 体系结构上的免费操作系统。这些黑客们开始贡献源代码，为他们特定的需要提供支持。

通常说，Linux 是一种 UNIX。从技术上说，Linux 是 UNIX 的克隆，因为它实现了 POSIX UNIX 规范 P1003.0。UNIX 从 1969 年诞生以来就统治着非 Intel 工作站领域，被公认为是一个强大而又优雅的操作系统。UNIX 既然定位在高性能工作站，就只在研究机构、学术单位以及开发机构中使用。Linux 把 UNIX 系统的能量带入了 Intel 个人计算机及其用户家里。如今，Linux 在工业和学术领域得到广泛应用，支持众多的体系结构，如 PowerPC。

本章简要介绍有关 Linux 的概念，引领你概览内核的组成和特性，并介绍一些引人入胜的 Linux 功能。为了理解 Linux 内核的概念，你需要对其开发动机有一个基本理解。

1.1 UNIX 的历史

刚刚说过，Linux 是一种 UNIX。尽管 Linux 并不是直接从现有的 UNIX 衍生而来的，但事实上，它实现了通用 UNIX 标准，因此有必要来看看 UNIX 的历史。

MULTICS (MULTiplexed Information and Computing Service) 被认为是 UNIX 操作系统的鼻祖，它是麻省理工学院、贝尔实验室和通用电气公司的一家合资企业开发的操作系统，该企业主要从事计算机制造。MULTICS 的开发是想让机器支持众多的分时用户。在合资企业成立的 1965 年，尽管当时操作系统支持多道程序设计（可在多个作业之间分时），但依然是只支持单用户的批处理系统。从用户提交一个作业到获得输出之间的响应时间要用小时计算。MULTICS 的主要目的是开发一个多用户分时系统，让每个用户都可以访问自己的终端。尽管贝尔实验室和通用电气公司最终放弃了这一项目，但 MULTICS 还是在许多地方得以实际应用。

UNIX 的开发始于移植精简的 MULTICS 版本，从而开发出一个 PDP-7 小型计算机上的操作系统，让这个新操作系统能支持一种新的文件系统，即 UNIX 文件系统的第一个版本。由 Ken Thompson 开发的这个操作系统，支持两个用户，有一个命令解释器和对新文件系统进行文件操作的一组程序。1970 年，UNIX 被移植到 PDP-11 上，经修改后能支持更多的用户。这就是第 1 版的 UNIX。

1973 年发布的 UNIX 第 4 版，由 Ken Thompson 和 Dennis Ritchie 用 C（由 Ritchie 刚刚开发的一种语言）重写。这就让操作系统脱离纯汇编语言，并打开操作系统可移植性的大门。想象一下这种转变的意义有多大！当时，操作系统完全是在系统的体系结构规范下建立起来的，因为汇编语言与体系结构密切相关，所以操作系统不易移植到其他体系结构。用 C 重写 UNIX 是向更可移植（和可读）的操作系统迈出的第一步，也是让 UNIX 迅速普及的一步。

1974 年是 UNIX 在大学中迅速普及的头一年。学术机构开始与贝尔实验室的 UNIX 系统开发组进行合作，开发出具有很多创新特色的第 5 版。这一版本可免费获得，其源代码可供大学用于教学。1979 年，在众多创新、代码简化以及改善可移植性之后，UNIX 操作系统第 7 版（V7）诞生。这一版本包含了 C 编译器和著名的 B shell 命令解释器。

20 世纪 80 年代出现了个人计算机。工作站当时只用在企业和大学中。大量 UNIX 变体从第 7 版衍生而来。这些变体包括 Berkeley UNIX (BSD) 和 AT&T UNIX System III 和 System V，其中 BSD 是由加利福尼亚大学伯克利分校开发的。每个变体又会演变出其他系统，如 NetBSD 和 OpenBSD (BSD 的变体) 以及 AIX (IBM 的 System V 变体)。事实上，UNIX 的所有商用变体都来源于 System V 或 BSD。

1991 年，Linux 出现，那时，UNIX 非常流行，但不适用于 PC。购买 UNIX 的价格令人望而却步，的确不是一般用户可以买的，除非他在大学工作。Linux 最初是作为 Minix 操作系统（由 Andrew Tanenbaum 开发的一个教学用小型操作系统）的扩展而实现的。

随后的几年，Linux 内核与 FSF (Free Software Foundation, 自由软件基金会) 的 GNU 项目所提供的系统软件相结合，使得 Linux^① 发展成为非常坚实的系统，吸引着贡献代码的黑客以外的

^① 为了体现 Linux 是 FSF 的 GNU 项目所提供的系统软件的组成部分，Linux 也被称为 GUN/Linux。

大量人员的注意。1994 年，Linux 的 1.0 版发布。从那时开始，Linux 迅速成长，发布版的需求量剧增，大批大学、公司及个人用户需要各种体系结构上的支持。

1.2 标准和通用接口

通用标准在不同变体的 UNIX 之间架起一座桥梁。用户选用哪种 UNIX 的决定影响到它的移植性，从而影响到它的潜在市场。如果你是一名程序开发者，很显然，你的程序能够拥有的市场只限于那些使用同一系统的人，除非你不辞辛苦地对该程序进行移植。标准的产生源于需要一种通用的编程接口规范，使得在一种操作系统上开发的程序在不修订或尽可能少修订的情况下可以在另一种操作系统上运行。各种标准组织都开始为 UNIX 制定规范。POSIX 就是其中的一种，这是由 IEEE 制定的一种用于计算机环境的可移植操作系统标准，Linux 就遵从这一标准。

1.3 自由软件和开放源码

Linux 是开放源码软件最成功的例子之一。所谓开放源码软件，就是它的源代码可以自由获取，这样任何人都可以修改、阅读和重新发布它。与之对立的是封闭源码软件，它仅以二进制形式发布。

开放源码允许用户按自己的意愿开发软件，以满足自己的需求。不过，根据许可证的内容，某些约束会对代码有限制。这样做的好处是用户决不受限于其他用户曾经开发的东西，因为他们可以自由地修改代码以满足自己的需要。任何人都可以对 Linux 操作系统进一步开发并贡献自己的智慧。这使得 Linux 的演进速度快得惊人，不管是开发、测试还是文档方面。

有各种各样的开源许可证存在：特别说明的是，Linux 是按 GNU 的 GPL (General Public License) 版本 2 的许可发布的。在源代码根目录的 COPYING 文件中，可以找到该许可证的副本。如果你打算修改 Linux 内核，那么最好熟悉这个许可证的条款，这样，你就可以知道自己贡献的合法权益。

关于自由和开放源码软件的产权有两大主要阵营。自由软件基金会 (Free Software Foundation) 和开源社区在意识形态方面有所不同。自由软件基金会是两大组织中较早出现的，他们坚持认为说软件是自由的，就像说言论是自由的一样。开源社区则认为自由而开放的软件是与专有软件并列的一种不同的方法论。更多的信息，请到 <http://www.fsf.org> 和 <http://www.opensource.org> 网站查阅。

1.4 Linux 发布版概览

我们曾提到，Linux 内核只是通常所说的“Linux”的一部分。Linux 发布版包含 Linux 内核、各种工具、窗口管理器以及很多其他应用程序。Linux 中使用的很多系统程序都是由 FSF GNU 项目开发和维护的。随着 Linux 日益流行和需求的不断增长，把内核和这些工具打包在一起成为一件意义重大且可从中获益的事。许多人和公司开始为一组特定的目标提供一个特定的发布版。我们不必关注过多的细节，只回顾一下到写本书为止的主要 Linux 发布版即可。新 Linux 发布版仍在继续发布着。

大多数 Linux 发布版都把工具和应用程序按头文件和可执行文件分组。这种分组就是所谓的包，使用 Linux 发布版就有这样的好处，不用自己去下载头文件并编译源代码。GPL 许可规定开源软件的增值部分是可以收费的，例如在代码重发布时提供的一些服务。

1.4.1 Debian

Debian^①是一种 GNU/Linux 操作系统。与其他发布版类似，大多数应用程序和工具都来自 GNU 软件和 Linux 内核。Debian 具有较好的包管理系统 apt (advanced packaging tool)。Debian 的主要缺点是最初的安装过程，会使 Linux 新手感到迷惑。Debian 并不属于某个公司，它是由自愿者社区开发的。

1.4.2 Red Hat/Fedora

Red Hat^② (公司) 是开源软件开发领域的大公司。Red Hat Linux 以前是该公司的 Linux 发布版，到了 2002~2003 年，它开始提供两种独立的发布版：Red Hat 企业级 Linux 和 Fedora Core。Red Hat 企业级 Linux 的目标是为企业、政府或者其他行业提供稳定并得到支持的 Linux 环境，而 Fedora Core 面向的是个人用户和爱好者。两个发布版之间的主要差异是一个比较稳定，一个新特性较多。相对于企业版，Fedora 会有较新但不太稳定的代码包含在发布版中。在美国，Red Hat 是 Linux 企业级版本的首选。

1.4.3 Mandriva

Mandriva Linux^③ (从前叫做 Mandrake Linux) 起源于 Red Hat Linux，是它的一种易安装版，但之后分枝为一个独立的发布版，该发布版面向的是 Linux 个人用户。Mandriva Linux 的主要特点是系统配置和安装都易上手。

1.4.4 SUSE

SUSE Linux^④ 是 Linux 平台的另一个主要参与者。SUSE 面向企业、政府、工业和个人用户。SUSE 的主要优势是它有一个安装和管理工具 Yast2。在欧洲，SUSE 是 Linux 企业级版本的首选。

1.4.5 Gentoo

Gentoo^⑤ 是新近上市的 Linux 发布版，它已经赢得众多赞许。Gentoo Linux 的主要特点就是它的所有包是根据自己机器的特殊配置从源代码编译而来的，这可以通过 Gentoo portage 系统来完成。

① <http://www.debian.org>

② <http://www.redhat.com>

③ <http://www.mandriva.com/>

④ <http://www.novell.com/linux/suse/>

⑤ <http://www.gentoo.org/>

1.4.6 Yellow Dog

Yellow Dog (黄狗) Linux^①是一种主要的基于 PowerPC 的 Linux 发布版。尽管刚才所说的很多发布版也都可以运行于 PowerPC 平台,但它们的重点还是基于 i386 的 Linux 版本。黄狗 Linux 非常类似于 Red Hat Linux,但进行了扩充开发,以支持较为普遍的 PowerPC 平台和基于 Apple 的专有硬件。

1.4.7 其他发布版

Linux 用户可以热情高昂地选择适合自己需要的发布版,而且选择余地很大。Slackware 是一款经典的发布版, MontaVista 更适用于嵌入式系统,当然,你还可以定制自己的发布版。要想进一步了解 Linux 各种各样的发布版,可以访问 Wikipedia 条目 http://en.wikipedia.org/wiki/Category:Linux_distributions 来查阅相关资料。

这里可能包含最新的信息,即使没有最新的信息,还可以通过链接访问其他站点上的信息。

1.5 内核版本信息

如果你想成为一名贡献者参与开发,与任何软件项目一样,理解项目的版本记录格式是很重要的。在 Linux 内核 2.6 版之前,开发社区遵循一种相当简单的版本和开发树编码方法。偶数版本 (2.2、2.4 和 2.6) 被认为是树的稳定分支。只有加入稳定分支的代码才是修订了错误的代码。内核开发会在标记为奇数 (2.1、2.3 和 2.5) 的开发树中继续进行。最终接受了大多数代码后,开发树几近完成,此时再发布一个新的稳定树。

在 2004 年中期,标准的发布周期有了一个变化:本该正常加入开发树的代码被包含进稳定的 2.6 树中。特别声明,“……主流内核将会是最快、特性最丰富的内核,但没必要是最稳定的。最终的稳定性是由发布者完成的(目前的确如此),但是,还是期望发布者能把他们的补丁快速融合进来。”(Jonathan Corbet 通过 <http://kerneltrap.org/node/view/3513> 发布。)

因为这是一个较新的开发模式,只有时间会证明这种发布周期今后是否会有重大变化。

1.6 基于 Power 的 Linux

基于 Power 的 Linux (Linux 系统运行在 Power 或者说 PowerPC 处理器上) 需求急增,变得日益流行。许多企业和公司购买基于 PowerPC 的系统来运行 Linux,其主要原因是 PowerPC 微处理器提供了极具扩展性的体系结构,能满足广泛的需求。

PowerPC 体系结构已经大举进入嵌入式市场, AMCC PowerPC 和 Motorola PowerPC 均推出了 32 位片上系统 (system-on-chip, SOC) 集成产品。这些 SOC 包含的东西除处理器外还有内置的时钟、内存、总线、控制器以及外设。

拥有 PowerPC 许可证的公司包括 AMCC、IBM 和 Motorola。尽管这三家公司都独立地开发自己的芯片,但是,这些芯片享有共同的指令集,相互之间是兼容的。

^① <http://www.yellowdoglinux.com/>

Linux 运行在基于 PowerPC 的游戏控制台、大型机以及桌面计算机上。Linux 在这日益主流的体系结构上的快速普及源于开源和私有企业积极主动的共同努力,前者如 <http://www.penguinppc.org>, 后者如 IBM 的 Linux 技术中心。

随着 Linux 在这一平台的日益流行,我们需要探究 Linux 如何与 PowerPC 进行交互并使用其功能。

许多站点包含基于 Power 的 Linux 的有用信息,当我们进行这种探索时会随时参考那些信息。<http://www.penguinppc.org> 就是追踪 Linux PPC 版本的站点,PowerPC 开发者社区也在这里了解基于 Power 的 Linux 的新闻。

1.7 什么是操作系统

现在我们考虑一下操作系统的一般概念、Linux 的基本用法与特点及它们之间如何联系起来。本节简要介绍一下这些概念,更详细的内容将在后续章节中介绍。如果你已经熟悉这些概念,就可以跳过本节,直奔第 2 章。

操作系统让裸机变为可用的计算机。它既负责管理系统硬件资源,也为应用程序的开发和执行提供基础。如果没有操作系统,每个程序就得为想要使用的所有硬件提供驱动程序,但那是应用程序开发者无法做到的。

操作系统的内部结构取决于其类型。Linux 和大多数 UNIX 的变体都是单模块系统。当我们说一个系统是单模块时,并不是指它是庞大无比的(尽管在大多数情况下,第二种解释也完全适用)。更确切地说,我们指它是由一个单独的模块单元——也就是一个单独的目标文件组成。操作系统是由很多例程经编译和链接在一起而形成的。例程之间交互的方式决定了单模块系统的内部结构。

在 Linux 中,我们把操作系统划分为内核空间和用户空间两部分。用户是通过用户空间与操作系统打交道的,程序员开发或使用的应用程序也位于用户空间。用户空间不能直接访问内核(从而不能访问硬件资源),但是可以通过内核定义的最外层例程——系统调用来访问。内核空间是硬件管理功能发挥作用的区域。在内核中,系统调用例程调用用户空间无法访问的内核例程来使用内核更细粒度的功能。

有一组例程对用户空间是不可见的,它们是各个设备驱动程序的函数和内核子系统的函数。设备驱动程序也提供了定义明确的接口函数,供系统调用或内核子系统访问。图 1-1 显示了 Linux 的结构。

Linux 还能提供动态可装载设备驱动程序,这弥补了单模块操作系统内在的一个主要缺陷。动态可装载驱动程序允许系统程序员把系统代码整合进内核中,而不是把代码编译进内核映像中。后者意味着较长时间的等待(具体时间取决于你系统的能力),并要重新启动,这极大地增加了系统程序员开发的用时。有了动态可装载设备驱动程序,系统程序员可以实时地装载和卸载他的设备驱动程序,无需重新编译整个内核,也无需重起系统。

Linux 的这些亮点“部分”会贯穿本书。我们尽可能遵循自顶向下的方法,以应用程序的例子开始,追踪其执行路径直到系统调用和内核子系统函数。通过这种方式,你可以把较为熟悉的

用户空间功能与对应的内核实现部分联系起来。

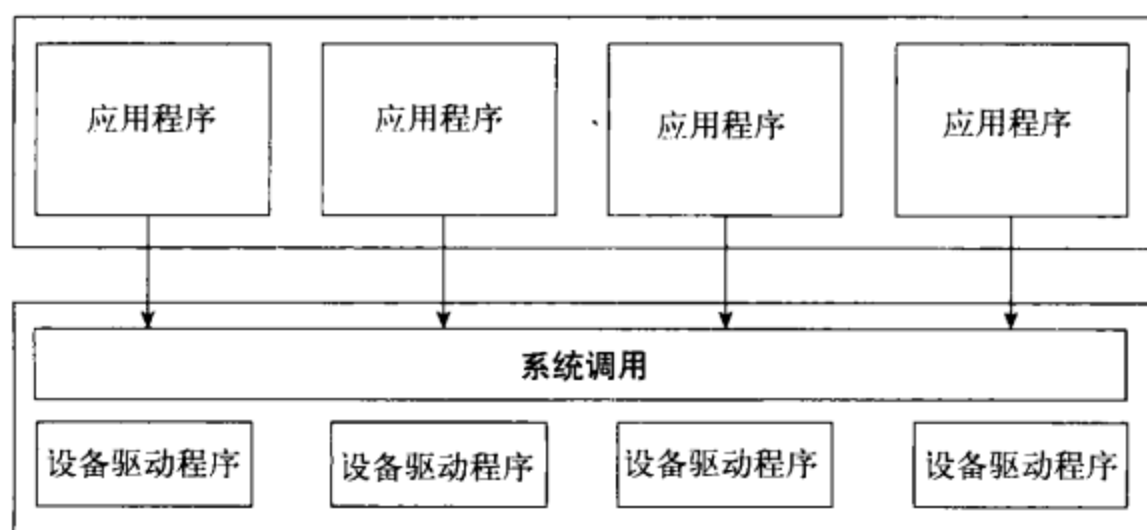


图 1-1 Linux 体系结构示意图

1.8 内核组织

Linux 支持多种体系结构，这就意味着，它可以运行在多种类型的处理器上，包括 alpha、arm、i386、ia64、ppc、ppc64 和 s390x。Linux 源代码包中有支持所有这些体系结构的代码。大多数源代码是用 C 写的，也是与硬件无关的。部分代码与硬件密切相关，是用 C 和特定体系结构的汇编语言写的。与机器密切相关的部分被封装成一系列系统调用，作为接口。你在本书中会读到，与体系结构相关的代码部分通常涉及系统初始化、启动，但不包括向量处理、地址转换和设备 I/O。

1.9 Linux 内核概述

Linux 内核中有各种各样的成分。在整本书中，我们交替使用术语成分（component）和子系统（subsystem），以强调内核函数的那些类别和功能上的差异。

在下一节，我们讨论其中的一些成分及它们在 Linux 内核中是如何实现的。我们还阐述操作系统的一些主要特点，以此来洞察其在内核中是怎样实现的。我们把操作系统的成分分为文件系统、进程、调度程序和设备驱动程序。尽管这并不是全面的划分，但它对本书其他部分的理解提供了参考。

1.9.1 用户接口

用户与系统之间通过程序进行交互。用户首先通过终端或虚拟终端登录到系统。Linux 中有一个程序，用于虚拟终端的叫做 **mingetty**，用于串行终端的叫做 **agetty**，它监控非活动终端，也就是监控等待用户发出登录请求的终端。登录时，他们需要输入自己的账户名，然后 **getty** 程序继续调用 **login** 程序，后者提示输入口令，为了进行认证，要访问账户名和口令列表，如果匹配，就允许用户进入系统，反之就退出并终止 **login** 进程。一旦该进程被终止，**getty** 程序全部重新复位，这就意味着即使该进程退出，它们还会重新启动。

得到系统的认证以后，用户需要通过一种方式告诉系统他们想做什么。如果用户成功得到认

证, login 程序就执行一个 shell。尽管从技术上说, shell 不是操作系统的组成部分, 但它是与操作系统进行交互的主要用户接口。shell 是一个命令解释器, 由一个监听进程组成。监听进程(一直处于阻塞状态, 直到接收输入的条件得到满足)解释并执行用户输入的请求。shell 位于图 1-1 中的顶层。

shell 显示命令提示符(通常是可配置的, 这取决于 shell), 并等待用户输入。用户通过输入符合 shell 语法的命令, 就可以与系统设备和程序进行交互。

用户能调用的程序是文件系统中他有执行权的可执行文件。对这些请求的执行是通过 shell 衍生的一个子进程进行的, 然后, 这个子进程会调用系统调用。系统调用返回后及子进程结束之后, shell 就可以回去继续监听用户的请求。

1.9.2 用户标识符

用户以唯一的账号名登录。不过, 他还可以关联一个唯一的用户 ID (UID)。内核用这个 UID 验证用户的文件访问权限。当一个用户登录后, 他就可以访问自己的主目录, 还可以在这个目录下创建、修改和删除文件。像 Linux 这样的多用户系统, 把用户与访问权限联系起来至关重要, 这可以限制或防止用户干预其他用户的活动或访问别人的数据。超级用户或 root 用户是没有权限限制的特殊用户, 它的用户 UID 为 0。

用户也是一个或多个组的成员, 每个组都有其唯一的组 ID (GID)。当一个用户被创建后, 他就会自动成为与他同名的那个组的成员。用户也可以被手动地加入到由系统管理员定义的其他组中。

一个文件或程序(可执行文件)总是与权限关联, 这些权限授予用户或组。任何特定的用户都可以决定谁可以访问他的文件, 谁不可以访问, 于是, 一个文件总是与某个特定的 UID 和某个特定的 GID 相关联。

1.9.3 文件和文件系统

文件系统提供了存储和组织数据的一种方法。Linux 把文件的概念描述为独立于设备的字节序列。通过这种抽象, 用户就可以访问不管存放在何种设备(例如硬盘、磁带驱动器和磁盘驱动器)上的文件。文件又被分组到叫做目录的容器中。因为目录可以相互嵌套(即一个目录可以包含另一个目录), 所以文件系统结构通常是一种层次树结构。树的根目录是顶层结点, 其他所有的目录和文件都存放在根下。根用斜杠(/)表示。文件系统存放在硬盘的一个分区或存储单元中。

1. 目录、文件和路径名

树中的每个文件都有一个标识其名称和位置的路径名。文件也有存放它的目录。从当前工作目录或用户所处的目录出发描述的目录就是所谓的相对路径, 因为文件的命名是相对于当前工作目录的。绝对路径是从文件系统的根开始的一个路径(例如, 从/开始的一个路径)。图 1-2 中用户 paul 的 file.c 的绝对路径是/home/paul/src/file.c。如果我们位于 paul 的主目录中, 相对路径则为 src/file.c。

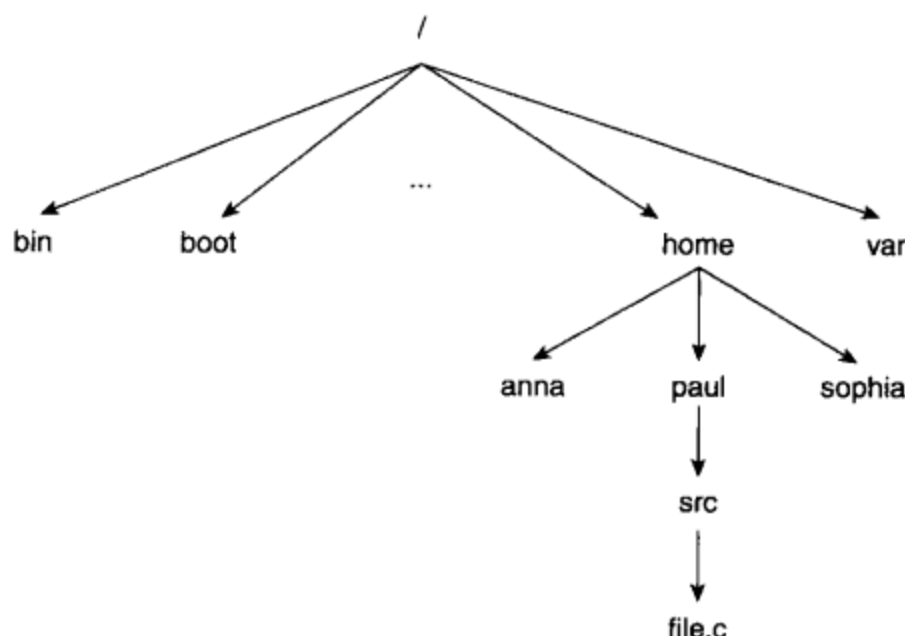


图 1-2 文件层次结构

之所以要有绝对路径和相对路径的概念，是因为内核把进程与当前工作路径和根目录相关联。当前工作路径就是进程被调用的地方，用一个点（.）表示。另外，父目录就是包含工作目录的目录，用两个点（..）表示。回想一下用户登录后，她“位于”自己的主目录中。如果 Anna 告诉 shell 执行某个特定的程序，例如 ls，只要她一登录，执行 ls 的进程就把/home/anna/作为它的当前工作目录（它的父目录是/home），而/将是它的根目录。根的父目录总是它本身。

2. 安装文件系统

Linux 与所有的类 UNIX 系统类似，文件系统只有被安装后才可以访问。文件系统的安装用 **mount** 系统调用，卸载用 **umount** 系统调用。文件系统安装在安装点上，安装点是一个目录，作为所安装的文件系统的根目录。安装点目录应当为空。作为安装点的目录，如果原来存放有任何文件，那么文件系统被安装后，这些文件都是不可访问的了，当然，文件系统被卸载后还是可以访问的。/etc/mtab 文件存放着所安装的文件系统表，而/etc/fstab 存放着列出所有文件系统及其属性的表。/etc/mtab 列出所安装的文件系统的设备，与之关联的安装点以及安装时的选项^①。

3. 文件保护和访问权限

文件的访问权限为文件提供了某种程度的私有性和安全性。访问权或许可权适用于三类不同的用户：用户本身、指定用户组及所有用户。这三种类型的用户被授予了不同的访问权限，所以对文件有三种访问权限：读、写和执行。当我们使用 ls-al 命令列出文件时，可以看到文件的权限：

```

lkp :~#- ls-al /home/sophia
drwxr-xr-x 22 sophia sophia    4096 Mar 14 15:13 .
drwxr-xr-x 24 root  root      4096 Mar 7 18:47 ..
drwxrwx--- 3 sophia department 4096 Mar 4 08:37 sources
  
```

第一行列出了 sophia 主目录的访问权限。她允许所有用户进入她的主目录，但不能编辑。她自己有读、写和执行的权限^②。第二行表示父目录/home 的访问权。/home 由 root 拥有，但是，

① 选项作为参数传递给 mount 系统调用。

② 执行权限，用于目录，表示用户可以进入；用于文件，表示用户可以执行文件，但只能用于可执行文件。

它允许所有用户都能读和执行。在 sophia 的主目录中，她有一个目录叫做 source，她允许自己和叫做 department 的组成员对这个目录有读、写和执行的权限，而其他用户没有任何权限。

4. 文件模式

除了访问权，文件还有另外三种模式：sticky、suid 和 sgid。让我们进一步查看每种模式。

● sticky

启用了 sticky 位的文件在 mode 字段的最后一个字符为“t”（例如，-rwx-----t）。回想一下磁盘访问比较慢的那个年代，内存不是很大，基于需求的方法论还没有被采纳^①，可执行文件可以启用 sticky 位，以确保内核可以让它待在内存，而不管它的执行状态。当把这一特性应用到一个频繁使用的程序时，就会提高性能，因为这减少了从磁盘访问文件信息所花费的时间。

当在一个目录上启用 sticky 位时，它防止对那个目录具有写权限的用户（不包括 root 用户和文件所有者）删除或更名其中的文件。

● suid

如果设置了 suid 位，那么用户访问权位上的可执行位“x”就变为“s”（例如，-rws-----）。当用户执行一个可执行文件时，执行进程就与调用这个文件的用户关联起来。如果可执行位上设置了 suid 位，那么进程就继承文件拥有者的 UID，因而就可以访问拥有者的所有访问权。这里引入了**实际用户 ID**的概念，这是与**有效用户 ID**相对的。当我们在 1.9.4 节考察进程时会看到，进程的实际 UID 对应着启动进程的用户的 UID，而有效 UID 通常与实际 UID 相同，除非在文件中设置了 setuid 位。如果设置了，则有效 UID 就具有了文件拥有者的 UID。

suid 曾被黑客使用，他们给 root 所拥有的可执行文件设置了 suid 位后调用它们，将程序的操作转去执行非 root 权限不能执行的指令。

● sgid

如果设置了 sgid 位，那么用户组访问权位上的可执行位“x”就变为“s”（例如，-rwxrws---）。sgid 位类似于 suid 位，只不过它用于组。进程也有**实际的组 ID**和**有效组 ID**，分别对应用户的 GID 和文件组的 GID。

5. 文件元数据

文件元数据就是对文件进行描述的所有信息，但不包括文件的内容。例如，元数据包括文件的类型、文件的大小、文件拥有者的 UID、访问权，等等。我们稍后会看到，某些文件类型（设备、管道及套接字）根本不包含任何数据，只有元数据。文件的所有元数据（不包括文件名）都存放在 inode 或称索引节点（index node）中。索引节点包含一组信息，每个文件都有自己的索引节点。**文件描述符**（file descriptor）是管理文件数据的一个内部核心数据结构，只有在进程访问文件时才可以获得文件描述符。

6. 文件类型

类 UNIX 系统有各种文件类型。

^① 这指的是，采用局部性原理装入程序块的技术，更多的细节参见第 4 章。

● 普通文件

普通文件 (regular file) 在模式字段中的第一个字符用破折号表示 (例如, `-rw-rw-rw-`)。普通文件可以包含 ASCII 数据, 也可以包含二进制数据 (如果是可执行文件)。内核并不关心存放在文件中的数据是什么类型, 因此对二者也不做什么区分。但是, 用户程序可能会关注。普通文件的数据存放在 0 个或者多个数据块中^①。

● 目录文件

目录 (directory) 文件在模式字段中的第一个字符用 “d” 表示 (例如, `drwx-----`)。目录也是一个文件, 其中存放文件名和文件索引结点之间的关联关系。目录是目录项组成的一个表, 其中每个表项对应目录下的一个文件。`ls -ai` 列出目录中的所有内容及其对应索引结点的 ID。

● 块设备文件

块设备 (block device) 在模式字段中的第一个字符用 “b” 表示 (例如, `brw-----`)。这些文件实际上表示硬件设备, 其 I/O 是以 2 的幂次方数据块大小进行传送的。块设备包括磁盘驱动器和磁带驱动器, 可以通过文件系统^②的 `/dev` 目录访问这些设备。访问磁盘比较耗时, 因此, 块设备的数据传送是通过内核的缓冲区缓存的, 这种暂存数据的方法减少了磁盘访问所花费的时间。在一定的时间间隔, 内核查看缓冲区中的数据是否被更新, 如果没有, 则同时把它刷新到磁盘。这极大地提高了性能。但是, 如果缓冲区中的数据还没有写到磁盘的话, 如果计算机发生崩溃就会导致缓冲区中数据的丢失。对磁盘驱动器的同步会强制调用 `sync`、`fsync` 或 `fdatasync` 系统调用, 这些调用负责把缓冲区中的数据写入磁盘。块设备文件只是一种表示, 并不使用任何数据块, 因为它不存放数据, 只需要一个索引结点来存放块设备的描述信息。

● 字符设备文件

字符设备 (character device) 在模式字段中的第一个字符用 “c” 表示 (例如, `crw-----`)。这些文件也表示硬件设备, 但数据不是以块组织的, I/O 是以字节流发送的, 并且在设备驱动程序和请求的进程之间直接传送。这些设备包括终端设备和串口设备, 在文件系统中通过 `/dev` 目录可以访问它们。有一种伪设备或者设备驱动程序也可以是字符设备, 它并不表示任何硬件, 而是执行了无关的内核方面的某些功能。这些设备也叫做原始设备, 因为没有中间缓存区存放数据。与块设备类似, 字符设备文件也不使用任何数据块, 因为它也不存放数据, 只需要一个索引结点来存放字符设备的描述信息。

● 链接文件

链接 (link) 设备在模式字段中的第一个字符用 “l” 表示 (例如, `lrw-----`)。链接就是指向文件的一个指针。这种文件类型允许某个特定的文件有多个引用, 但实际上在文件系统中只有一份文件的副本, 也只有一份数据。有两种类型的链接: 硬链接 (hard link) 和符号链接 (symbolic link), 后者也叫软链接 (soft link)。这两种类型的链接都是通过调用 `ln` 创建的。硬链接有一些限制, 但软链接没有。这些限制包括链接的文件必须是同一文件系统的, 不能链接到目录, 不能链接不存在的文件。链接文件的权限就是它所指向的文件的权限。

① 空文件有 0 个数据块。

② mount 系统调用需要一个块文件。——译者注

- 命名管道文件

管道 (pipe) 文件在模式字段中的第一个字符用“p”表示 (例如, `prw-----`)。管道是一个文件, 它作为数据管道方便程序之间的通信: 一个程序把数据写入管道, 另一个程序从其中读出。管道实际上缓存了来自第一个进程的输入数据。命名管道也称作 FIFO, 这是因为这种管道以先进先出的方式向读程序传递信息。与设备文件非常类似, 管道文件也没有数据块, 仅有索引结点。

- 套接字文件

套接字 (socket) 在模式字段中的第一个字符用“s”表示 (例如, `srw-----`)。套接字是方便进程之间通信的特殊文件。套接字和管道的一个不同在于, 套接字能促使通过网络连接的不同计算机的进程之间进行通信。套接字文件也不与任何数据块关联。因为本书不涵盖联网的内容, 我们就不详细介绍套接字的内在机理了。

7. 文件系统的类型

Linux 文件系统为各种文件系统类型共存提供了一种接口。文件系统类型的决定因素有块数据被分割的方式、在物理设备上操作的方式及物理设备的类型。文件系统类型的例子包括网络文件系统, 例如 NFS, 磁盘文件系统, 例如 ext3, 这是 Linux 默认的文件系统。某些特殊的文件系统, 例如 `/proc`, 提供对内核数据和地址空间的访问。

8. 文件控制权

访问 Linux 中的文件时, 控制权的传递经过好几个阶段。首先, 要访问文件的程序发出一个系统调用, 例如 `open()`、`read()` 或 `write()`。然后, 控制权传递给执行系统调用的内核。有一个叫做 VFS 的文件系统高级抽象, 它决定文件所存在的具体文件系统类型 (例如, ext2、minix 还是 msdos), 之后, 控制权就传递给相应的文件系统驱动程序。

文件系统驱动程序根据给定的逻辑设备处理文件的管理工作。硬盘驱动器可能有 msdos 和 ext2 分区。文件系统驱动程序知道如何对存放在设备上的数据进行解释, 也知道对与文件相关的所有元数据进行记录。因此, 文件系统驱动程序存储实际的文件数据和附加的信息, 例如, 时间戳、组和用户模式, 以及文件权限 (`read/write/execute`)。

然后, 文件系统驱动程序调用低级的设备驱动程序处理, 从设备读取数据的操作。这种低级设备驱动程序知道有关块、扇区及所有的硬件信息, 这些是在设备上存取数据块所必要的信息。低级设备驱动程序把数据上传给文件系统设备驱动程序, 驱动程序又解释和格式化这些原始数据, 并把信息传递给 VFS, VFS 最终把数据传回原始的调用程序。

1.9.4 进程

如果说操作系统是开发者所依赖的框架, 那么, 进程就是由这个框架所负责和管理的基本活动单元。更确切地说, 进程就是正在执行的一个程序。一个程序可以执行多次, 因此, 一个特定的程序就可能对应多个进程。

随着 20 世纪 60 年代多用户操作系统的引入, 进程的概念变得越来越重要。在单用户操作系统中, CPU 只执行一个进程, 当前运行的进程还没有完成之前, 其他进程都不能执行。引入多用户操作系统后 (或者我们希望并发执行多任务后), 我们需要定义任务之间切换的方式。

进程模型通过定义**执行上下文** (execution context) 让多个任务可以并发执行。在 Linux 中, 每个进程运行起来就好像只有它一个进程存在。操作系统根据预定义的一组规则把处理器分配给这个或那个进程, 以此来管理这些上下文。**调度程序** (scheduler) 定义并执行这些规则。调度程序跟踪进程运行及切换所花费的时间, 以确保任何进程都不能贪婪地占有 CPU。

执行上下文由对应程序的相关部分组成, 包括程序的数据 (以及它能访问的内存地址空间)、寄存器、栈及栈指针、程序计数器的值等。除了进程的数据和内存寻址, 进程的其他部分都不需要程序员操心。不过, 栈、栈指针、程序计数器以及机器寄存器需要操作系统来管理。在多进程系统中, 操作系统还必须肩负起进程之间**上下文切换**的重任, 并管理进程对系统资源的竞争。

1. 进程创建和控制

一个进程是通过另一个进程创建的, 这是通过调用 **fork()** 系统调用实现的。当进程调用 **fork()** 时, 我们说调用进程**产生**了或**分叉**出一个新进程。新进程就是**子进程**, 而原进程就是**父进程**。所有进程都有一个父亲, **init** 进程除外。所有进程都是第一个进程 **init** 的后代, **init** 是在系统启动时就产生的, 这将在下一节进一步讨论。

作为这种父/子模式的结果, 系统就有了一棵**进程树**, 它定义了所有运行进程之间的关系, 图 1-3 展示了一棵进程树。

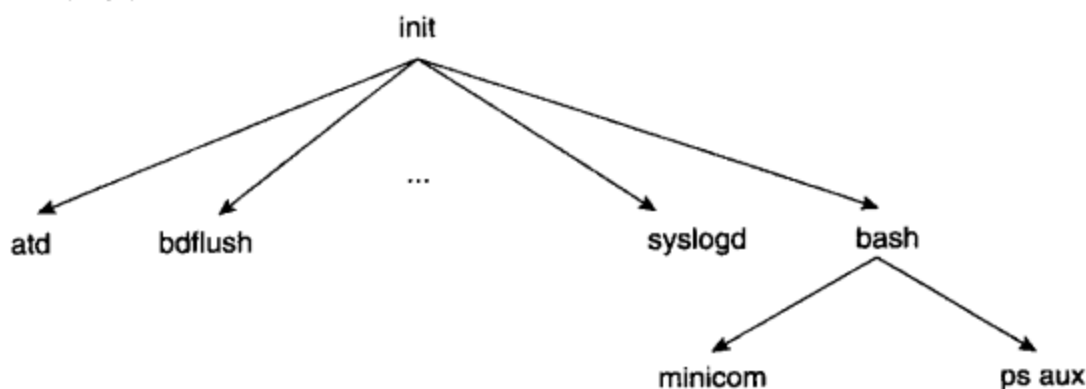


图 1-3 进程树

当子进程被创建时, 父进程可能想知道子进程什么时候能完成。**wait()** 系统调用用来暂停父进程, 直到子进程退出。

一个进程也可能用另一个进程替代它自己。前面描述的 **mingetty()** 函数就是一个例子。当用户发出请求要访问系统时, **mingetty()** 函数要求用户输入他的名字, 然后, 用执行 **login()** 的进程替代自己, 当然, 用户名作为参数传递过去。这种替代是通过调用一个 **exec()** 类系统调用实现的。

2. 进程 ID

每个进程都有一个唯一的标识符, 叫做**进程 ID (PID)**。PID 是一个非负整数。当进程被创建时, 就以递增的顺序给其分配进程 ID。当到达最大 PID 值时, 又从头开始, 把大于 1 的最小可用值分配给 PID。有两个特殊的进程:**进程 0** 和**进程 1**。进程 0 负责系统初始化, 并创建进程 1, 进程 1 也叫做 **init** 进程。在 Linux 系统中运行的所有进程都是进程 1 的后代。进程 0 执行以后, **init** 进程变得无所事事了。8.5 节将讨论这一进程。

有两个系统调用可以识别进程。**getpid()** 系统调用获得当前进程的 PID, 而 **getppid()** 系统调用获得进程的父进程的 PID。

3. 进程组

进程可以是进程组中的一名成员，一组中的所有成员具有相同的组ID。进程组把相关的进程组成一组，有时你需要这么做，例如，你想杀死一组无所事事的进程，可以让它们成为一组，同时接收到kill信号。如果进程的PID与组ID相同，则该进程是组长。可以对进程组ID进行操作，调用getpgid()获得指定进程的进程组ID，而调用setpgid()设置其进程组ID。

4. 进程状态

进程可以处于不同的状态，这一方面依赖于调度程序，另一方面依赖于进程所争用的系统资源是否可用。如果进程处于运行状态，说明它正在被运行，或者处于运行队列中，这是一种保存队列中等待执行的进程引用的数据结构。如果进程处于睡眠状态，说明它等待资源或让出处理机给其他进程；如果进程处于死亡状态，说明进程已经被杀死；如果进程已经结束，但其父进程没有调用它的wait()，则该进程成为僵尸进程。

5. 进程描述符

每个进程都有一个进程描述符，存放描述进程的所有信息。进程描述符包含的信息有：进程状态、PID、启动进程所用的命令，等等。这些信息可以通过调用ps (process status) 命令来显示。调用ps可能产生如下的结果：

```
lkp:~#ps aux | more
USER PID TTY STAT COMMAND
root 1 ? S init [3]
root 2 ? SN [ksoftirqd/0]
...
root 10 ? S< [aio/0]
...
root 2026 ? Ss /sbin/syslogd -a /var/lib/ntp/dev/log
root 2029 ? Ss /sbin/klogd -c 1 -2-x
...
root 3324 tty2 Ss+ /sbin/mingetty tty2
root 3325 tty3 Ss+ /sbin/mingetty tty3
root 3326 tty4 Ss+ /sbin/mingetty tty4
root 3327 tty5 Ss+ /sbin/mingetty tty5
root 3328 tty6 Ss+ /sbin/mingetty tty6
root 3329 ttyS0 Ss+ /sbin/agetty -L 9600 ttyS0 vt102
root 14914 ? Ss sshd: root@pts/0
...
root 14917 pts/0 Ss -bash
root 17682 pts/0 R+ ps aux
root 17683 pts/0 R+ more
```

以上列出的进程信息表明，PID为1的进程是init进程，mingetty()和agetty()程序分别在虚拟终端和串口终端进行监听。请注意，它们都是init进程的子进程。最后，结果还显示了发出ps aux | more命令的bash会话。注意，用来表示管道的|本身不是一个进程。前面我们说管道方便了两个进程之间的通信，这里两个进程分别是ps aux和more。

另外，STAT列表示进程的状态，其中S表示睡眠进程，R表示正在运行或可运行的进程。

6. 进程优先级

在单处理器的计算机中，我们一次只能执行一个进程。当进程之间互相争用执行时间时，就

给它们分派不同的优先级。这个优先级由内核动态改变，改变的依据是进程已经运行的时间以及在这个时间点上优先级的取值。给进程分派一个时间片（timeslice）让其运行，时间片到后，它就被换出，由调度程序选择另一个进程运行，随后我们将进一步描述。

高优先级进程首先运行，运行的次数也较多。用户可以调用 `nice()` 系统调用设置进程的优先级。这个调用表示一个进程对另一个进程友好的程度，意味着进程到底愿意让出多少优先级。高优先级值为负，而低优先级值为正。我们设的 `nice` 值越大，表明该进程越愿意转让运行权。

1.9.5 系统调用

系统调用是用户程序与内核通信的主要机制。系统调用通常被封装在库调用中，这些调用对每个系统调用执行前所需的寄存器的设置和数据进行管理。然后，用户程序把合适的例程链接在库中以对内核发出请求。

系统调用通常会应用于具体的子系统。这就意味着用户空间的程序可以通过这些系统调用与任一特定的内核子系统进行交互。例如，文件有文件处理的系统调用，而进程有针对具体进程的系统调用。贯穿于本书，我们所说的系统调用总是与特定的内核子系统相关联。例如，当我们谈及文件系统时，就考察 `read()`、`write()`、`open()` 和 `close()` 系统调用。这就可以让你了解文件系统在内核中是如何实现与管理的。

1.9.6 Linux 调度程序

把控制权从一个进程移交给另一个进程，是 Linux 调度程序的主要职责。任何进程（包括 Linux 2.6 中的内核抢占在内）都可以被随时打断，并把控制权传递给一个新进程。

例如，当中断发生时，Linux 必须停止执行当前进程并处理中断。另外，像 Linux 这样的多任务操作系统，必须确保没有进程能贪婪地占有 CPU。调度程序处理两种任务：一方面，用新进程换出当前进程；另一方面，记录进程的 CPU 使用情况，指示如果它们运行太长时间就应当被换出。

第 7 章将深入阐述 Linux 调度程序如何决定哪个进程获得 CPU 的控制权。简要地说，调度程序对优先级的确定根据两个方面，一是过去的执行情况（进程曾占用多长时间的 CPU），二是进程的紧要程度（中断比登录进程更紧要）。

Linux 调度程序还管理进程在多处理器机器（SMP）上的执行方式。在多个处理器上做负载均衡，把进程与具体的 CPU 绑定在一起，这都需要一些有意思的特性。但总的来说，各个 CPU 基本的调度功能是相同的。

1.9.7 Linux 设备驱动程序

设备驱动程序是内核与硬盘、内存、声卡、网卡以及很多其他的输入输出设备进行交互的接口。

Linux 内核通常在默认的安装中包含很多这样的驱动程序。如果说你不能通过键盘输入任何数据，那么 Linux 还有多大用处呢？设备驱动程序被封装在一个模块中。Linux 尽管是一个单模块内核，但达到了高度的模块化，每个设备驱动程序都可以动态地安装。因此，默认的内核可以保持相对较小，然后根据承载 Linux 运行的系统实际配置逐渐扩充。

在 Linux 2.6 内核中，系统用户可以用两种方式查看设备驱动程序的状态：`/proc` 和 `/sys` 文件系统。简单地说，`/proc` 通常用来调试和监控设备，而 `/sys` 用来改变设置。例如，如果在嵌入式 Linux 设备中你有一台射频调谐器 (RF tuner)，在 `sysfs` 设备目录项下，默认的调谐器频率不仅是可查看的，也是可以改变的。

在第 5 章和第 10 章中，我们会进一步介绍字符设备和块设备的设备驱动程序。更具体地说，我们将介绍 `/dev/random` 设备驱动程序，并了解在 Linux 系统中它怎样从其他设备收集熵信息。

1.10 可移植性和体系结构的相关性

当我们探索 Linux 内核内在机理时，不可避免地要讨论到底层硬件或体系结构的某些方面。毕竟，Linux 内核是运行在某种具体处理机上的庞大软件，因此，它势必涉及某种处理器指令集和处理能力的详细信息。然而，这并不需要每个内核或系统程序员成为宿主微处理器的专家，不过，充分了解内核代码如何组织或分层有助于程序员解决所碰到的棘手问题。

Linux 内核经过巧妙设计，使其代码尽可能少地直接依赖底层硬件。当需要与硬件交互时，就在编译时引入合适的库以执行特定于体系结构的那个专门函数。例如，当内核想进行上下文切换时，就调用 `switch_to()` 函数。因为内核是针对特定体系结构（例如，PowerPC 或 x86）编译的，所以它就把合适的 `include` 文件 `include/asm-ppc/system.h` 或 `include/asm-i386/system.h`（在编译时）链接进来，这个头文件中就包含了 `switch_to()` 与体系结构相关的定义。在启动时，与体系结构相关的初始化代码调用 Firmware 或者 BIOS (BIOS 是系统启动软件，将在第 9 章讨论)。

有一层软件直接与硬件打交道，到底是哪一层软件，取决于目标体系结构。在这层之上，内核代码就好像遗忘了底层硬件。

因为这个原因，可以说，Linux 内核是跨不同体系结构可移植的。当驱动程序不可移植性时，就产生了某种局限，或者是因为该硬件对某些体系结构不可用，或者是因为没有足够的移植需求。为了开发一个设备驱动程序，程序员必须了解给定硬件设备的寄存器级规范。并不是所有的厂家都愿意提供这种文档，因为硬件是他们专有的。这也间接地限制了 Linux 跨体系结构的可移植性。

1.11 小结

这一章引入本书要详细展开的内容，并给予简要概述。我们也提及了致使 Linux 如此流行的一些特性，以及围绕这一操作系统存在的某些问题。下一章将介绍有效探索 Linux 内核所需的基本工具。

1.12 习题

1. Unix 系统和 Unix 的克隆系统之间有何区别？
2. 术语“基于 Power 的 Linux”指什么？
3. 什么是用户空间？什么是内核空间？
4. 用户空间的程序访问内核功能的接口是什么？
5. 用户 UID 与用户名有何联系？

6. 列举文件与用户关联的方式。
7. 列举 Linux 所支持的各种文件类型。
8. shell 是操作系统的一部分吗？
9. 为什么既要有文件保护还要有文件模式？
10. 列出在存放文件元数据的结构中都有哪类信息。
11. 字符设备和块设备的根本区别是什么？
12. 是哪个组成部分让 Linux 内核成为多进程系统？
13. 一个进程如何成为另一个进程的父进程？
14. 本章中，我们介绍了两种层次树：文件树和进程树。它们之间有何相同点，又有何不同点？
15. 进程 ID 被关联到用户 ID 吗？
16. 分配进程优先级有何作用？所有用户能改变进程的优先级值吗？为什么？
17. 设备驱动程序仅用于添加硬件支持吗？
18. 是什么原因使 Linux 能够在不同体系结构间移植？



第 2 章

内核探索工具集

本章内容

- 内核中常见的数据类型
- 汇编
- 汇编语言示例
- 内联汇编
- 特殊的 C 语言用法
- 内核探索工具一览
- 内核发言：倾听来自内核的消息
- 其他奥秘

本章简要介绍 Linux 中一般的编码结构，并描述诸多与内核打交道的方法。首先关注的是 Linux 中用于高效存储和信息检索的常见数据类型、编码方法及基本汇编语言，这将为在后面的章节中详细分析内核打下基础。接下来介绍 Linux 如何将源代码编译、链接成可执行代码，以利于理解跨平台编码和更好地介绍 GNU 工具集。然后简要介绍从 Linux 内核搜集信息的多种方法。本章涉及的内容非常广泛，包括源代码和可执行代码的分析，以及如何在 Linux 内核中插入调试语句。最后，本章以“大杂烩”的形式总结并评论 Linux 中其他常见的习惯用法^①。

2.1 内核中常见的数据类型

Linux 内核中包含许多对象和数据结构，例如内存页面、进程和中断。如果操作系统要高效运行，那么如何及时地从多个对象中引用其中某个对象将是至关重要的。Linux 使用链表和二叉搜索树（以及一组辅助例程）先将这些对象分组放入一个容器中，然后再以某种有效的方式查找单个元素。

2.1.1 链表

在计算机科学中，**链表**是一种常见的数据类型，广泛用于 Linux 内核中。它在 Linux 内核中

^① 我们尚未深入研究内核的本质，此处仅总结了在内核代码中遨游所必需的工具和基本概念。如果你是一个有经验的内核研究者，可以跳过这一节，直接从第 3 章开始深入研究内核。

常以循环双向链表的形式出现（参见图 2-1）。因此，给定链表中的任一节点，都可以找到其下一节点和上一节点。链表的完整代码存放在头文件 `include/linux/list.h` 中。本节讨论链表的主要特征。

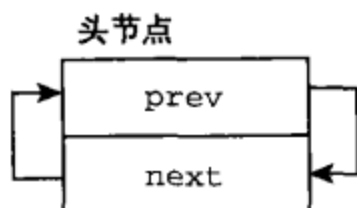


图 2-1 调用宏 `INIT_LIST_HEAD` 后的链表

以下是使用宏 `LIST_HEAD` 和 `INIT_LIST_HEAD` 初始化链表的代码：

```
-----
include/linux/list.h
27
28 struct list_head {
29     struct list_head *next, *prev;
30 };
31
32 #define LIST_HEAD_INIT(name) { &(amp;name), &(amp;name) }
33
34 #define LIST_HEAD(name) \
35     struct list_head name = LIST_HEAD_INIT(name)
36
37 #define INIT_LIST_HEAD(ptr) do { \
38     (ptr)->next = (ptr); (ptr)->prev = (ptr); \
39 } while (0)
-----
```

第 34 行：宏 `LIST_HEAD` 根据给定的名字 `name` 创建链表的表头。

第 37 行：宏 `INIT_LIST_HEAD` 将表头节点中的 `prev` 指针和 `next` 指针都初始化为指向表头节点本身，完成这两个宏调用后，`name` 就指向一个空的双向链表^①。

相应地，简单的栈和队列也可以由函数 `list_add()` 或 `list_add_tail()` 分别来实现，工作队列的代码中给出了一个典型的例子：

```
-----
kernel/workqueue.c
330 list_add(&wq->list, &workqueues);
-----
```

内核将 `wq->list` 加入到系统的工作队列链表 `workqueues` 中，因此 `workqueues` 就是一个栈形式的队列。

与此类似，下列代码将 `work->entry` 添加到链表 `cwq->worklist` 的末尾，`cwq->worklist` 因而也被当作队列：

```
-----
kernel/workqueue.c
84 list_add_tail(&work->entry, &cwq->worklist);
-----
```

① 空链表的定义：其头节点的 `next` 指针指向该链表的表头元素本身。

从链表中删除某个元素可以调用函数 `list_del()`。该函数将要删除的链表元素作为参数，删除元素时，仅需修改该元素的下一个节点和前一个节点，使之互相指向对方即可。例如，当销毁一个工作队列时，下列代码可以从系统的工作队列链表中删除该工作队列：

```
-----
kernel/workqueue.c
382 list_del(&wq->list);
-----
```

`include/linux/list.h` 中定义了一个特别有用的宏 `list_for_each_entry`：

```
-----
include/linux/list.h
349 /**
350 * list_for_each_entry - iterate over list of given type
351 * @pos: the type * to use as a loop counter.
352 * @head: the head for your list.
353 * @member: the name of the list_struct within the struct.
354 */
355 #define list_for_each_entry(pos, head, member)
356     for (pos = list_entry((head)->next, typeof(*pos), member),
357         prefetch(pos->member.next);
358         &pos->member != (head);
359         pos = list_entry(pos->member.next, typeof(*pos), member),
360         prefetch(pos->member.next))
-----
```

该函数循环遍历整个链表，操作链表中的每个元素。例如，当 CPU 工作时，将为每个工作队列唤醒一个进程：

```
-----
kernel/workqueue.c
59 struct workqueue_struct {
60     struct cpu_workqueue_struct cpu_wq[NR_CPUS];
61     const char *name;
62     struct list_head list; /* Empty if single thread */
63 };
...
466 case CPU_ONLINE:
467     /* Kick off worker threads. */
468     list_for_each_entry(wq, &workqueues, list)
469         wake_up_process(wq->cpu_wq[hotcpu].thread);
470     break;
-----
```

该宏展开并使用 `workqueue_struct wq` 中的 `list_head list` 链表来遍历那些头节点位于工作队列中的链表。如果这看起来让人有点困惑的话，请记住，为了遍历链表并不需要知道我们是哪个链表的成员。当前节点的 `next` 指针指向链表的头节点^①时，就已经访问到该链表的表尾了。有关工作队列的说明参见图 2-2。

① 我们也可以通过 `list_for_each_entry_reverse()` 来反向遍历链表。

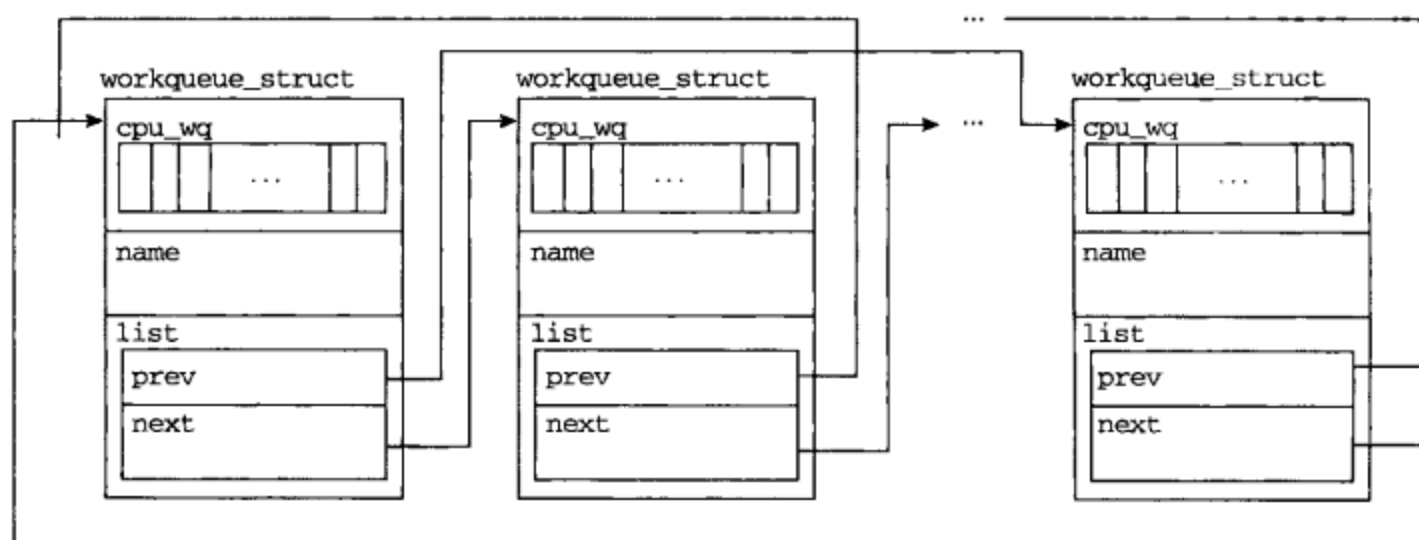


图 2-2 工作队列链表

与在前一节中讨论过的带有双指针的头节点相反，这里我们还可以修改链表，使其头节点中仅有一个指向第一个元素的指针，这样的头节点应用于散列表（参见第 4 章），它没有指向链表表尾元素的指针。由于在散列查找中不常用到尾指针，因而这样做可以节省内存空间。

```
-----
include/linux/list.h
484 struct hlist_head {
485     struct hlist_node *first;
486 };

488 struct hlist_node {
489     struct hlist_node *next, **pprev;
490 };

492 #define HLIST_HEAD_INIT { .first = NULL }
493 #define HLIST_HEAD(name) struct hlist_head name = { .first = NULL }
-----
```

第 492 行：宏 HLIST_HEAD_INIT 将指针 first 置为空指针。

第 493 行：宏 HLIST_HEAD 根据名称创建链表，并将指针 first 置为空指针。

正如我们将会在调度程序、定时器和模块处理例程（module-handling routine）中看到的那样，整个 Linux 内核的工作队列代码中都用到了这些链表结构。

2.1.2 查找

上一节我们分析了链表中的分组元素。有序表根据链表中每个元素的关键值进行排序（例如，每个元素的关键值均大于其前一元素中对应的值）。如果想要找到某个特定元素（基于其关键值），可以从表头开始，顺序查找整个链表，比较当前节点的关键值与给定的关键值。如果不相等，就继续比较下一个元素，直到找到匹配的值为止。采用这种查找方法时，从链表中查找给定元素所花的时间与其关键值成正比。换句话说，如果增加链表中元素个数，这种线性查找将花费更多时间。

大O表示法

对于查找算法而言，大O表示法常用于从理论上衡量一个算法找到某给定关键值的执行时间，它代表对于一个给定值 n 在最坏情况下所花费的查找时间。线性查找的效率是 $O(n/2)$ ，这就意味着，平均来算，找到一个给定关键值必须与链表中的一半元素进行比较。

出处：美国标准和技术协会 (www.nist.gov)

对于元素较多的链表而言，为了不使操作系统空等，需要更快地存储和定位给定数据的方法。虽然目前已经实现了很多方法（包括其派生方法），但 Linux 存储数据时用的另一主要数据结构还是树。

2.1.3 树

树用于 Linux 内存管理中，能够高效地访问并操作数据。此时，其效率就是用存储和检索数据的快慢程度来衡量的。本节将讨论基本树，重点介绍红黑树，而关于树在 Linux 下的具体实现及其辅助例程，请参考第 6 章。在计算机科学中，有根树由节点和边组成（参见图 2-3），节点代表数据元素，边代表节点之间的路径。第一个节点，或者说顶层节点，就是有根树的根节点。节点之间的相互关系有父、子、兄弟这三种。每个子节点有且仅有一个父节点（根节点除外），每个父节点可以有一个或多个子节点，互为兄弟的节点有共同的父节点，没有子的节点称为叶节点。树的高度是指从树根到最远的叶节点之间的边数。树的每一行子孙被称为一层。在图 2-3 中，b 和 c 位于 a 的下一层，而 d、e、f 位于 a 下面两层。查找给定兄弟节点集合中的某一数据元素时，有序树最左边的兄弟节点其值最小，而最右边的兄弟节点其值最大。树通常以链表和数组的形式来实现，而在树中移动的过程就叫树的遍历。

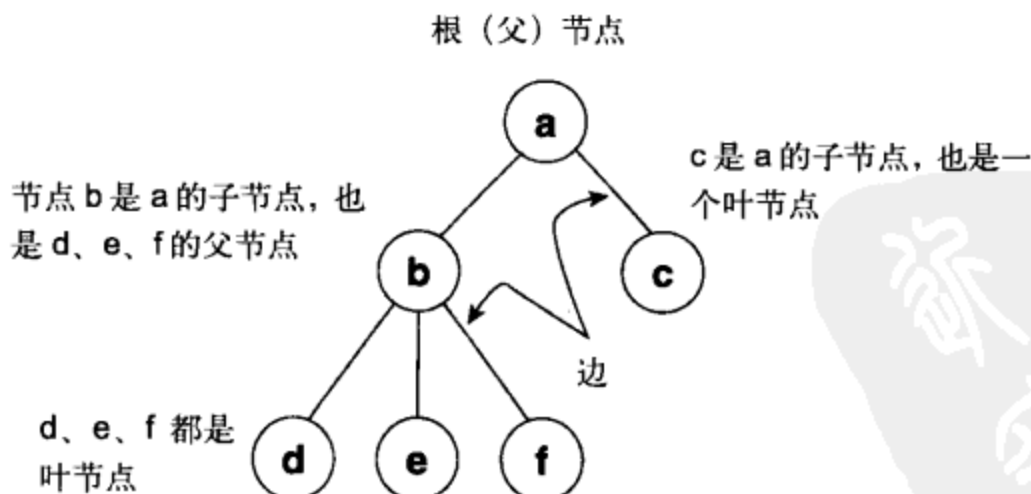


图 2-3 有根树

1. 二叉树

前面我们用线性查找的方式来查找关键值，在每次循环中比较关键值的大小。如果每次比较都可以将有序表中待处理的节点数目减半呢？

二叉树和链表不同，它是一种非线性的分层数据结构。在二叉树中，每个元素或节点指向一个左子或右子节点，每个子节点又指向它的一个左子或右子节点，以此类推。其节点之间排序的

主要规则就是左子节点的关键值小于父节点的关键值，而右子节点的关键值大于或等于父节点的关键值。因此，对于一个给定节点的关键值，其左子树上所有节点的关键值都小于该节点，而其右子树上所有节点的关键值都大于或等于该节点。

往二叉树中存放数据时，首先必须找到适当的插入位置，而每次循环都可以将要查找的数据个数减半。用大 O 表示法来表示时，其性能（关于查找的次数）就是 $O(\log(n))$ ，相比之下，线性查找的性能是 $O(n/2)$ 。

遍历二叉树的算法比较简单。对于每个节点而言，比较完该节点的关键值后，就可以遍历其左子树或者右子树，因而，二叉树的遍历本身就很方便用递归来实现。下面将讨论其具体实现、辅助函数以及二叉树的类型。

刚才我们提到，二叉树中的节点可以有一个左子节点，或者一个右子节点，或者有左、右两个子节点，也可以没有子节点。有序二叉树的规则是，给定一个节点的值 x ，其左子节点（包括所有子孙节点）的值小于 x ，而其右子节点（包括所有子孙节点）的值大于 x 。由此可知，如果将数据的有序集合插入到二叉树中，将形成一个线性列表，对于一个给定值，其查找速度就会变得和线性查找一样慢。例如，根据数据集 $[0,1,2,3,4,5,6]$ 创建一颗二叉树时，0 是树根；1 比 0 大，是 0 的右子节点；2 比 1 大，是 1 的右子节点；而 3 是 2 的右子节点，以此类推。

在均高二叉树中，根节点到任意叶节点的距离都是一样远的。节点添加到二叉树中后，为了保证查找的效率，必须进行平衡化处理，这可以通过旋转来实现。插入一个节点后，给定节点 e ，如果它有一个比任何其他叶节点高两层的左子树，就必须右旋转 e 。如图 2-4 所示， e 变成 h 的父节点， e 的右子节点则变成 h 的左子节点。如果每次插入节点后都进行了平衡化处理，那么，每次最多只需要作一次旋转。满足平衡规则（若某节点的子节点是一个叶节点的话，它们之间的间距不会超过 1）的二叉树被称为 AVL 树（这一术语最初是由 G.M.Adelson-Velskii 和 E.M.Landis 提出来的）。

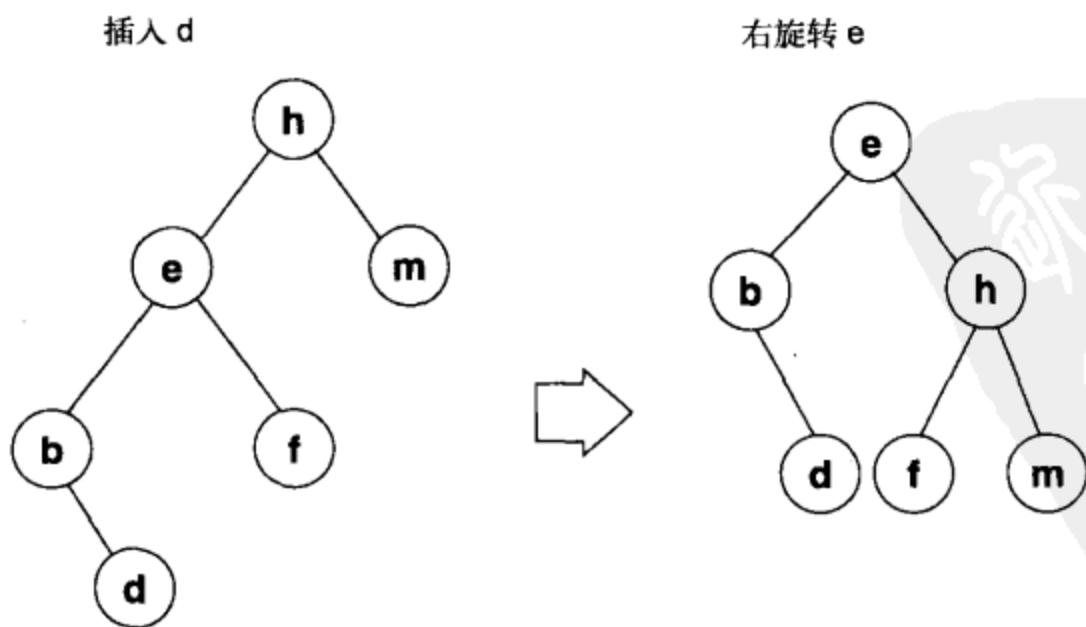


图 2-4 二叉树的右旋转

2. 红黑树

红黑树类似于 AVL 树，用于 Linux 内存管理。红黑树就是平衡二叉树，其每个节点都有红或

黑的颜色属性。

红黑树的规则如下：

- 每个节点要么是红色的，要么是黑色的；
- 如果一个节点是红色的，那么它的所有子节点都是黑色的；
- 所有叶节点都是黑色的；
- 从根节点到叶节点的每条路径包含同样多的黑色节点。

AVL 树和红黑树的查找效率都是 $O(\log(n))$ ，而根据插入（已排序的/未排序的）和查找数据的不同，每次都能得出不同的具体数值。（网上有一些讨论二叉搜索树[BST]性能的文章，有兴趣的话，可以找来看看。）

前面已经提到，许多其他数据结构和相关的查找算法也被应用于计算机科学中。本节的主要目的是，希望通过介绍 Linux 内核中常用数据结构的基本概念，帮助探索 Linux 内核。对链表和树结构有一定了解后，就可以更好地理解复杂的操作，例如将在后续章节讨论的内存管理和队列。

2.2 汇编

作为一个操作系统，Linux 的某些部分不可避免地与其运行它的处理器密切相关。当然，Linux 设计者们已经做了大量工作，使得与处理器（或“体系结构”）相关的代码尽可能少一些，可以在支持的所有体系结构中重用的代码尽可能多一些。本节探讨如下内容：

- 用 C 语言编写的同一个函数在 x86 和 PowerPC 体系结构中分别是如何实现的；
- 宏与内联汇编代码的应用。

本节旨在提供足够的基础知识，以便研究特定于体系结构的内核代码时，能很好地理解它，而不至于手足无措。有关汇编语言程序设计的更多内容，请参阅其他书籍。此外，我们还将介绍一些与体系结构相关的最复杂的代码：内联汇编。

在讨论 PPC 与 x86 汇编语言之前，我们还是先来看看这两种处理器的体系结构。

2.2.1 PowerPC

PowerPC 是 RISC（精简指令集计算机）体系结构。RISC 体系结构设计目的是提高系统的性能，这通过一个能在尽可能少的处理器周期中执行的简单指令集来实现。读者很快就会看到，由于利用了硬件可执行并行指令（超标量体系结构）特性，某些指令非常复杂。IBM、摩托罗拉和苹果公司联合定义了 PowerPC 体系结构。表 2-1 列出了 PowerPC 中可供用户使用的寄存器。

表2-1 PowerPC的用户寄存器集合

寄存器名	不同体系结构下的位宽		功 能	寄存器数量
	32位	64位		
CR	32	32	条件寄存器	1
LR	32	64	链接寄存器	1
CTR	32	64	计数寄存器	1
GPR[0..31]	32	64	通用寄存器	32

(续)

寄存器名	不同体系结构下的位宽		功 能	寄存器数量
	32位	64位		
XER	32	64	定点异常寄存器	1
FPR[0..31]	64	64	浮点寄存器	32
FPSCR	32	64	浮点状态控制寄存器	1

表 2-2 说明了通用寄存器和浮点寄存器在 ABI (Application Binary Interface, 应用程序二进制接口) 方面的应用。可变寄存器任何时候都可以使用, 专用寄存器用于特定场合, 非可变寄存器使用时必须在整个函数调用过程中加以保护。

表2-2 ABI寄存器的用法

寄 存 器	类 型	使 用
r0	可变	在函数开始或者结束时使用, 与具体语言相关
r1	专用	栈指针
r2	专用	内容表指针TOC
r3-r4	可变	参数传递, 作为第一个参数和返回地址
r5-r10	可变	参数传递, 作为函数或系统调用开始的参数
r11	可变	用于指针的调用和当作一些语言的环境指针
r12	可变	用于异常处理和glink (动态链接器) 代码中
r13	非可变	必须在调用过程中加以保护
r14-r31	非可变	必须在调用过程中加以保护
f0	可变	擦除
f1	可变	第一个浮点参数, 第一个浮点型标量返回值
f2-f4	可变	第2~4个浮点参数, 浮点型标量返回值
f5-f13	可变	第5~13个浮点参数
f14-f31	非可变	必须在调用过程中加以保护

ABI

ABI 是一些规范的集合, 例如调用规则、机器接口、操作系统接口等。它允许链接程序将已经编译好的单个模块链接成一个整体, 而不需要重新编译。ABI 定义了这些单元之间的二进制接口。目前, PowerPC 有几个 ABI 的变体, 它们通常与目标操作系统和硬件相关。这些变体或补充物有基于 UNIX System V 应用二进制接口的文档, 最初源自 AT&T, 后来又根据 SCO (Santa Cruz Operation) 修改而成。遵循 ABI 的好处在于: 可以链接不同编译器所编译的目标文件。

32 位 PowerPC 体系结构采用 4 字节长、按字对齐的指令, 存取时按字节、半字、字、双字来操作, 可分为分支指令、定点指令和浮点指令这几类。

1. 分支指令

CR (Condition Register, 条件寄存器) 对所有分支操作而言都是不可缺少的一部分, 它可以

划分为 8 个 4 位的区域。数据移动指令（例如 `mtcrf` 指令）可以显式设置 CR 的值，某些指令执行后也能隐式的修改 CR 的内容，例如最常见的比较指令。

LR（Link Register，链接寄存器）由特定形式的分支指令使用，以提供目标地址和返回分支程序执行完后的地址。

CTR（CounT Register，计数寄存器）存储由特殊的分支指令执行的减 1 循环计数。CTR 也可以保存某些特定的分支指令的目标地址。

除上述 CTR 和 LR 外，PowerPC 的分支指令还可以跳转到某一相对或绝对地址，通过扩展的助记符，可以产生多种形式的条件分支和无条件分支。

2. 定点指令

PPC 中没有修改存储器内容的运算指令，对存储单元的所有操作都必须事先将数据加载到一个或多个 32 位通用寄存器（GPR）中。存储器存取指令按字节、半字、字、双字以大端序来存取数据。使用扩展的助记符后，增加了许多加载、存储、算术运算和逻辑运算的定点指令，以及移入/移出系统寄存器的特殊指令。

3. 浮点指令

浮点指令可分为两类：一类是运算指令，包括算术运算指令、取整指令、转换指令和比较指令；另一类是非运算指令，包括在存储器与寄存器之间或寄存器与寄存器之间传送数据的指令。系统中共有 32 个通用浮点寄存器，均可用于存放双精度浮点数。

大端模式/小端模式

在处理器体系结构中，端序是指字节顺序和操作。PowerPC 采用大端模式，就是说，最高位字节存储在低地址空间，最低位字节存储在随后的 3 个地址空间中（对 32 位数据而言）。小端模式用于 x86 体系结构，与大端模式刚好相反，最低位字节位于低位地址空间，而最高位字节位于随后的 3 个地址空间中。让我们来看看 0x12345678 的表示（参见图 2-5）。

32 位大端模式（PPC）

12	34	56	78
0	7 8	15 16	23 24 31

32 位小端模式（x86）

78	56	34	12
0	7 8	15 16	23 24 31

图 2-5 大端模式和小端模式

本书并不讨论哪个系统采用的方式更好一些，但编写和调试代码时，了解你是在哪种系统上工作是极为重要的。这有一个错用端序的例子：为基于一种体系结构的 PCI 编写使用另一体系结构的设备驱动程序。

术语大端模式/小端模式源自 Jonathan Swift 撰写的《格利佛游记》。在这个故事中，格利佛发现了因吃熟鸡蛋的方式完全不同而敌对的两个民族，一个从大端开始，另一个从小端开始。

2.2.2 x86

x86 体系结构是一种 CISC（复杂指令集计算）体系结构。根据其功能的不同，指令的长度是可变的。奔腾系列的 x86 体系结构有三种寄存器：通用寄存器、段寄存器和状态/控制寄存器。

8 个通用寄存器及其用法如下所示。

- **EAX**: 通用累加器。
- **EBX**: 指向数据的指针。
- **ECX**: 用于循环操作的计数器。
- **EDX**: I/O 指针。
- **ESI**: 指向 DS 段中数据的指针。
- **EDI**: 指向 ES 段中数据的指针。
- **ESP**: 栈指针。
- **EBP**: 指向栈中数据的指针。

以下 6 个段寄存器用于实模式中的寻址操作。此时按块访问内存，因而内存中某个给定的字节可通过该段中的偏移量来引用（例如，ES:EDI 引用的是 ES（附加段）中的某个单元，其偏移量存储在寄存器 EDI 中）。

- **CS**: 代码段。
- **SS**: 栈段。
- **ES、DS、FS、GS**: 数据段。

EFLAGS 寄存器存放每条指令执行完后处理器的状态，它可以保存 0、溢出（overflow）、进位（carry）等结果。EIP 是专用的指针寄存器，存储处理器当前指令的偏移量，与代码段寄存器配合使用，以形成一个完整的地址（如，CS:EIP）。

- **EFLAGS**: 状态、控制和系统标志。
- **EIP**: 指令指针，包含 CS 段中的偏移量。

x86 体系结构中的数据顺序遵循小端模式，访问内存时可按字节（8 位）、字（16 位）、双字（32 位）和四字（64 位）访问。地址转换（及其相关的寄存器）见第 4 章，本节中只需要知道在 x86 体系结构中，存储代码和数据操作指令的常用寄存器可以分为三类：控制寄存器、算术运算寄存器和数据传送寄存器。

1. 程序控制指令

与 PPC 中的分支指令类似，控制指令可以改变程序的流程。x86 体系结构使用了各种各样的跳转指令和标号，使之能够根据 EFLAGS 寄存器的值选择要执行的代码。跳转指令有多种变体，表 2-3 列举出了其中最常用的一些变体。条件代码要根据特定指令的执行结果来设置。例如，对两个整型操作数执行 **cmp**（比较）指令后，将修改 EFLAGS 寄存器的如下标志位：**OF**（溢出）、**SF**（非负标志）、**ZF**（零标志）、**PF**（奇偶标志）、**CF**（进位标志）。因此，如果 **cmp** 指令比较的是两个相等的操作数，零标志将被设置。

表2-3 常见的跳转指令

指 令	作 用	EFLAGS条件代码
je	相等时跳转	ZF=1
jg	大于时跳转	ZF=0, SF=OF
jge	大于等于时跳转	SF=OF
jl	小于时跳转	SF!=OF
jle	小于等于时跳转	ZF=1
jmp	无条件跳转	无条件

在 x86 汇编代码中, 标号由唯一的名字后加冒号组成。它可以出现在汇编程序的任何地方, 并与紧跟其后的那行代码具有相同的地址。下列代码中使用了条件跳转和标号:

```
-----
100  pop eax
101  loop2:
102  pop ebx
103  cmp eax, ebx
104  jge loop2
-----
```

第 100 行: 取出栈顶元素并将其值存入 `eax` 中。

第 101 行: 名为 `loop2` 的标号。

第 102 行: 取出栈顶元素并将其值存入 `ebx` 中。

第 103 行: 比较 `eax` 与 `ebx` 的值。

第 104 行: 如果 `eax` 的值大于或等于 `ebx` 的值则跳转。

另一种转移程序控制的方法是使用 `call` 和 `ret` 指令。参见下面这行汇编代码:

```
-----
call my_routine
-----
```

这条 `call` 指令转移控制程序到 `my_routine` 标号处, 同时将其下一条指令的地址推入栈, 然后, `ret` 指令 (在 `my_routine` 中执行) 将弹出返回地址, 并跳转到该位置。

2. 算术运算指令

常见的算术运算指令有 `add`、`sub`、`imul` (整数乘)、`idiv` (整数除), 以及 `and`、`or`、`not`、`xor` 等逻辑运算符。

本书并不讨论 x86 的浮点指令及其相关寄存器。近来, Intel 和 AMD 体系结构的扩展, 如 MMX、SSE、3DNow、SIMD、SSE2/3 等, 大大增强了运算量较大的数学应用, 例如图形和声音方面的应用。相关体系结构其详情可查阅编程手册。

3. 数据传送指令

数据可以在寄存器之间、寄存器和内存之间传送, 也可以将常量存入寄存器或内存, 但是不能从内存直接传送到内存。请看如下示例:

```
-----
100  mov eax, ebx
-----
```

```

101  mov eax,WORD PTR[data3]
102  mov BYTE PTR[char1],al
103  mov eax,0xbeef
104  mov WORD PTR [my_data],0xbeef

```

第 100 行：将 ebx 中的值（32 位）移动到 eax 中。

第 101 行：将内存变量 data3 的值（32 位）移动到 eax 中。

第 102 行：将内存变量 char1 的值（8 位）移动到 al 中。

第 103 行：将常数 0xbeef 移动到 eax 中。

第 104 行：将常数 0xbeef 传入内存变量 my_data 中。

如上例所示，**push**、**pop**、**pushl**、**popl** 都可以将数据移入/移出栈（由 SS:ESP 来定位）。与 mov 指令类似，push 和 pop 指令也可以操作寄存器、内存单元中的数据和常数。

2.3 汇编语言示例

我们可以创建一个简单的程序，来看看对于同样的 C 语言代码，不同体系结构下究竟是怎样来生成汇编语言代码的。本次试验使用的是 Red Hat 9 自带的 gcc 编译器和用于 PowerPC 的 gcc 交叉编译器。我们给出了 C 程序，并对照给出了 x86 下和 PowerPC 下的汇编代码。

看到寥寥几行 C 语言代码产生了多少汇编代码后，读者也许会大吃一惊。我们仅将 C 语言代码编译成汇编语言代码，没有链接 C 运行时库或者创建/撤销局部栈等任何环境代码，因此其大小当然比实际的 ELF 可执行程序小很多。

值得注意的是，在汇编程序中，可以确切地看到处理器在周而复始地运行过程中获取了什么指令。当然，另一种查看的方法是，你也可以完全控制你的代码和系统。值得一提的是：即使指令是从内存中顺序取出来的，它们的执行顺序也不是每次都和读入顺序完全一样，在某些体系结构中，顺序加载和存储操作是相互独立的。

以下是该例子的 C 语言代码：

```

-----
count.c
1 int main()
2 {
3   int i,j=0;
4
5   for(i=0;i<8;i++)
6     j=j+i;
7
8   return 0;
9 }
-----

```

第 1 行：main 函数的定义。

第 3 行：将局部变量 i、j 初始化为 0。

第 5 行：for 循环：当 i 从 0 到 7 取值时，使得 j=j+i。

第 8 行：return 表示跳转回调用程序。

2.3.1 x86 中的汇编示例

这是 x86 中在命令行内键入 `gcc -S count.c` 后产生的代码。阅读代码前，应当知道：栈的基址由 `SS:ebp` 给出；代码按“AT&T”格式生成；寄存器前加`%`前缀，常数前加`$`前缀。读完本节前面给出的汇编指令示例后，你应该能读懂这个简单的程序了，但进一步研究之前，还要讨论一个间接寻址的变量。

当涉及内存中的某个位置时（比如栈），汇编程序用特殊的语法来表示索引寻址。基址寄存器放在圆括号内，而索引（或偏移量）放在圆括号外，有效地址就是索引值加上基址寄存器的值。例如，若`%ebp`的值是 20，则`-8(%ebp)`的有效地址就是 $(-8) + (20) = 12$ 。

```
-----
count.s
1  .file "count.c"
2  .version "01.01"
3  gcc2_compiled.:
4  .text
5  .align 4
6  .globl main
7  .type main,@function
8 main:
   # 为 i 和 j 创建 8 字节的局部内存区
9  pushl %ebp
10 movl %esp, %ebp
11 subl $8, %esp

   # 将 i（对应的内存单元为 ebp-4）和 j（对应的内存单元为 ebp-8）初始化为 0
12 movl $0, -8(%ebp)
13 movl $0, -4(%ebp)
14 .p2align 2
15 .L3:

   # 这是一个用于测试的 for 循环
16 cmpl $7, -4(%ebp)
17 jle .L6
18 jmp .L4
19 .p2align 2
20 .L6:

   # 这是 for 循环的循环体
21 movl -4(%ebp), %eax
22 leal -8(%ebp), %edx
23 addl %eax, (%edx)
24 leal -4(%ebp), %eax
25 incl (%eax)
26 jmp .L3
27 .p2align 2
28 .L4:

   # 设置该函数的退出代码
29 movl $0, %eax  30 leave  31 ret
-----
```

第 9 行：将栈基址指针推入栈。
 第 10 行：将栈指针移入基址指针。
 第 11 行：从 `ebp` 开始，分配 8 字节的栈 `mem`。
 第 12 行：将 0 存入地址为 `ebp-8` 的内存单元（即将变量 `j` 赋值为 0）。
 第 13 行：将 0 存入地址为 `ebp-4` 的内存单元（即将变量 `i` 赋值为 0）。
 第 14 行：这是一个汇编指令，表明该指令按半字对齐。
 第 15 行：这是由汇编程序创建的名为 `.L3` 的标号。
 第 16 行：该指令比较 `i` 和 7 的大小。
 第 17 行：如果 `i` 的值小于等于 7，则跳转到标号 `.L6` 处。
 第 18 行：否则，跳转到 `.L4` 处。
 第 19 行：对齐。
 第 20 行：标号 `.L6`。
 第 21 行：将 `i` 的值移入 `eax`。
 第 22 行：将 `j` 的地址加载到 `edx`。
 第 23 行：将 `i` 的值与 `edx` 所指向的存储单元的内容相加，即将 `i` 与 `j` 的值相加，结果放在 `i` 中。
 第 24 行：将 `i` 的新值移入 `eax`。
 第 25 行：`i` 增加 1。
 第 26 行：跳转到 `for` 循环处，继续进行下一次循环。
 第 27 行：按照第 14 行的注释所描述的方式对齐。
 第 28 行：标号 `.L4`。
 第 29 行：在 `eax` 中设置返回代码。
 第 30 行：释放局部内存区。
 第 31 行：从栈中弹出变量的值和返回地址，跳转回调用程序。

2.3.2 PowerPC 中的汇编示例

以下是根据 C 程序产生的 PPC 汇编代码。如果你熟悉汇编语言（及其简写），对许多 PPC 指令的功能就十分清楚了。但是，我们还得讨论几个基本指令的派生形式。

- **stwu RS, D(RA)** (Store Word with Update)：这一指令取出通用寄存器 (GPR) `RS` 的值，并存储到有效地址为 `RA + D` 的内存单元。之后，用新的有效地址来更新通用寄存器 (GPR) `RA` 的值。
- **li RT, RS, SI** (Load Immediate)：这是定点加载指令的扩展助记符号，等价于将 `RS`、`SI` 的值相加，其中通用寄存器 `RS` 和 `SI` 的和存储于 `RT` 中，`RS` 和 `SI` 是两个 16 位补码。若 `RS` 是 (GPR) `R0`，则 `SI` 的值存储在 `RT` 中。要注意的是，只有 16 位的数值必须做如下操作，即操作码、寄存器和数值都必须重新编码成长度为 32 位的指令。
- **lwz RT, D(RA)** (Load Word and Zero)：这条指令和 `stwu` 一样，生成一个有效地址，并从内存中加载一个字的数据到通用寄存器 `RT` 中。“and Zero” (Load Word and Zero) 标

志是指当 64 位地址运行于 32 位模式时, 计算后的有效地址的高 32 位会被置为 0。(详情参见 *PowerPC Architecture Book I* 一书)。

□ **blr** (Branch to Link Register): 这条指令无条件转移到存储在链接寄存器中的 32 位地址。调用函数时, 调用程序将返回地址存入链接寄存器。与 x86 中的 `ret` 指令类似, `blr` 是从函数返回的常用方式。

以下代码是从命令行键入 `gcc -S count.c` 后生成的:

```
-----
countppc.s
1  .file "count.c"
2  .section ".text"
3  .align 2
4  .globl main
5  .type main,@function
6  main:
    #从栈空间创建 32 字节的内存区, 并初始化 i 和 j.
7  stwu 1,-32(1) #Store stack ptr(r1)32 bytes into the stack
8  stw 31,28(1) #Store word r31 into lower end of memory area
9  mr 31,1 #Move contents of r1 into r31
10 li 0,0 #Load 0 into r0
11 stw 0,12(31) #Store word r0 into effective address 12(r31), var j
12 li 0,0 #load 0 into r0
13 stw 0,8(31) #Store word r0 into effective address 8(r31), var i
14 .L2:
    #for 循环的测试部分
15 lwz 0,8(31) #Load i into r0
16 cmpwi 0,0,7 #compare word immediate r0 with integer value 7
17 ble 0,.L5 #Branch if less than or equal to label .L5
18 b .L3 #Branch unconditional to label .L3
19 .L5:
    #for 循环的循环体
20 lwz 9,12(31) #Load j into r9
21 lwz 0,8(31) #Load i into r0
22 add 0,9,0 #Add r0 to r9 and put result in r0
23 stw 0,12(31) #Store r0 into j
24 lwz 9,8(31) #load i into r9
25 addi 0,9,1 #Add 1 to r9 and store in r0
26 stw 0,8(31) #Store r0 into i
27 b .L2
28 .L3:
29 li 0,0 #Load 0 into r0
30 mr 3,0 #move r0 to r3
31 lwz 11,0(1) #load r1 into r11
32 lwz 31,-4(11) #Restore r31
33 mr 1,11 #Restore r1
34 blr #Branch to Link Register contents
-----
```

第 7 行: 将栈指针 `ptr` (即寄存器 `r1`) 的 32 字节压入栈。

第 8 行: 将寄存器 `r31` 的值存储到内存区的低地址空间。

第 9 行: 将 `r1` 的值赋给 `r31`。

第 10 行: 设置寄存器 `r0` 的初值为 0。

第 11 行: 将 `r0` 的值存储到有效地址为 $(r31+12)$ 的地址空间, 即用寄存器 `r0` 的值为变量

j 赋值。

第 12 行：设置寄存器 r0 的值为 0。

第 13 行：将 r0 的值存储到有效地址为 (r31+8) 的地址空间，即用寄存器 r0 的值为变量 i 赋值。

第 14 行：标号.L2:。

第 15 行：将 i 赋给寄存器 r0。

第 16 行：比较寄存器 r0 中的值与整数 7 的大小。

第 17 行：r0 小于或等于 7 时跳转到标号.L5 处。

第 18 行：无条件跳转到标号.L3 处。

第 19 行：标号.L5。

第 20 行：将 j 赋给寄存器 r9。

第 21 行：将 i 赋给寄存器 r0。

第 22 行：r0 加上 r9，其结果存储在 r0 中。

第 23 行：将寄存器 r0 的值存储到变量 j 中。

第 24 行：将 i 赋给寄存器 r9。

第 25 行：r9 加 1，其结果存储在 r0 中。

第 26 行：将寄存器 r0 的值存储到变量 i 中。

第 27 行：无条件跳转到标号.L2 处。

第 28 行：标号.L3:。

第 29 行：将 0 赋给寄存器 r0。

第 30 行：将 r0 的值传给 r3。

第 31 行：将 r1 的值赋给 r11。

第 32 行：恢复寄存器 r31 的内容。

第 33 行：恢复寄存器 r1 的内容。

第 34 行：跳转到链接寄存器所存储的地址处。

对照这两个汇编程序文件，发现它们的行数几乎是一样的，进一步观察可以看出，RISC (PPC) 处理器使用了许多加载和存储指令，而 CISC (x86) 更偏好于使用 mov 指令。

2.4 内联汇编

gcc 编译器支持的另一种编码形式是内联汇编 (inline assembly) 代码。名如其言，内联汇编不需要调用单独编译好的汇编程序。我们可以通过特定的结构告诉编译器将代码块组合到一起，而不是要编译该代码块。虽然这样做会生成与体系结构相关的文件，但能大大提高 C 函数的可读性和执行效率。

以下就是内联汇编程序的结构：

```
-----
1  asm (assembler instruction(s)
2    : output operands    (optional)
```

```

3   : input operands    (optional)
4   : clobbered registers (optional)
5   );
-----

```

例如，内联汇编程序最基本的形式是：

```
asm ("movl %eax, %ebx");
```

也可以写成

```
asm ("movl %eax, %ebx" : : : );
```

由于确实要修改 `ebx` 的内容，我们逐渐揭开了编译器神秘的面纱。

能够接受并修改 C 表达式的值，然后将它们返回给程序，同时确信编译器知道这些变化，正是这种能力使得内联汇编技艺超群。让我们进一步来探索其参数传递的奥秘吧。

2.4.1 输出操作数

第 2 行，冒号后就是输出操作数，它是一个 C 表达式列表，其后的圆括号中是约束条件。对输出操作数而言，约束条件通常用 `=` 修饰符来修饰，表示“只写”的意思；修饰符 `&` 表示这是一个已被修改过的操作数，说明该指令使用这个操作数之前，它已经被修改过了。操作数之间要用逗号分开。

2.4.2 输入操作数

除了没有只写修饰符 `=` 外，第 3 行的输入操作数遵循与输出操作数相同的语法。

2.4.3 已修改过的寄存器（已修改的元素列表）

我们可以在汇编语句中修改各种各样的寄存器和内存单元的内容，将它们列举出来，可以方便 `gcc` 知晓这些内容已经被修改过了。

2.4.4 参数的编号方式

每个参数都给定一个位置编号，所有参数从 0 开始统一编号。例如，如果有一个输出参数和两个输入参数，那么，`%0` 就是输出参数，`%1` 和 `%2` 分别对应一个输入参数。

2.4.5 约束条件

约束条件指明怎样使用操作数。GNU 文档中列出了所有简单的约束和与平台相关的约束。表 2-4 列举了 x86 最常用的约束。

表2-4 x86的简单约束和与平台相关的约束

约 束	作 用
a	寄存器 <code>eax</code>
b	寄存器 <code>ebx</code>

(续)

约 束	作 用
c	寄存器ecx
d	寄存器edx
s	寄存器esi
D	寄存器edi
I	常数 (0~31)
q	从eax、ebx、ecx、edx中动态分配一个寄存器
r	与q+esi, edi一样
m	内存定位
A	与a+b的作用相同。同时分配eax和ebx, 形成一个64位寄存器

2.4.6 asm

实际上 (尤其是在 Linux 内核中), 由于与其他结构同名, 关键字 **asm** 也许会在编译时引发错误。读者常常会看到写成 `__asm__` 的表达式, 两者完全是一回事。

2.4.7 __volatile__

另一个常用的修饰符是 `__volatile__`, 该修饰符对汇编代码的意义非同寻常, 它告诉编译器不要优化这个内联汇编例程。通常, 随着硬件的软件化, 编译器认为我们资源丰富, 浪费成性, 于是尝试重写代码, 使之尽可能高效。这对应用程序设计十分有用, 但在硬件级程序设计中将会适得其反。

例如, 假定我们正向一个由变量 **reg** 表示的、指向某一存储单元的寄存器中写数据, 接下来初始化某些需要轮询 **reg** 的动作。编译器仅把这当做同一存储单元的连续读操作, 并去掉了明显冗余的部分。使用修饰符 `__volatile__` 后, 编译器就知道不要优化使用了该变量的存取操作。同样, 当你看到内联汇编代码块中出现 `asm volatile(...)` 字样时, 编译器也不会优化这一代码块。

现在, 我们基本了解了汇编和 gcc 内联汇编, 可以来看一看真正的内联汇编代码了。首先运用所学知识来分析一个简单的例子, 之后才是稍复杂一些的代码块。

下面是第一个例子的代码, 它将变量传给一个内联汇编代码块:

```

-----
6 int foo(void)
7 {
8 int ee = 0x4000, ce = 0x8000, reg;
9 __asm__ __volatile__ ("movl %1, %%eax";
10 "movl %2, %%ebx";
11 "call setbits" ;
12 "movl %%eax, %0"
13 : "=r" (reg) // reg [param %0] is output
14 : "r" (ce), "r"(ee) // ce [param %1], ee [param %2] are inputs
15 : "%eax" , "%ebx" // %eax and % ebx got clobbered

```

```

16  )
17  printf("reg=%x",reg);
18  }

```

第6行：本行是C例程的开始。

第8行：将局部变量 ee、ce、reg 作为参数传给内联汇编程序。

第9行：本行是内联汇编例程的开始，它将 ce 的值传给 eax。

第10行：将 ee 的值传给 ebx。

第11行：在汇编程序中调用函数。

第12行：将返回值存储到寄存器 eax 中，并将其复制给变量 reg。

第13行：本行是输出参数列表，参数 reg 的属性为只写。

第14行：本行是输入参数列表，ce 和 ee 是寄存器变量。

第15行：本行是寄存器修改列表。该例程改变了寄存器 eax 和 ebx 的值，编译器知道在执行完这个例程后不要使用 eax 和 ebx 的值。

第16行：本行标志着内联汇编例程的结束。

第二个例子是函数 switch_to() 的应用。该函数在头文件 include/asm-i386/system.h 中定义，是 Linux 上下文切换的核心部分。本章仅探讨它的内联汇编机制，第9章将详细分析如何使用 switch_to()。

```

-----
include/asm-i386/system.h
012 extern struct task_struct * FASTCALL(__switch_to(struct task_struct *prev,
struct task_struct *next));
...
015 #define switch_to(prev,next,last) do {
016     unsigned long esi,edi;
017     asm volatile("pushfl\n\t"
018         "pushl %%ebp\n\t"
019         "movl %%esp,%0\n\t" /* save ESP */
020         "movl %5,%%esp\n\t" /* restore ESP */
021         "movl $1f,%1\n\t" /* save EIP */
022         "pushl %6\n\t" /* restore EIP */
023         "jmp __switch_to\n\t"
023         "1:\n\t"
024         "popl %%ebp\n\t"
025         "popfl"
026         : "=m" (prev->thread.esp), "=m" (prev->thread.eip),
027         "=a" (last), "=S" (esi), "=D" (edi)
028         : "m" (next->thread.esp), "m" (next->thread.eip),
029         "2" (prev), "d" (next));
030 } while (0)
-----

```

第12行：FASTCALL 告诉编译器将参数传给寄存器。

asm linkage 标志则告诉编译器将参数存入栈。

第15行：do{statements...}while(0) 这样的编码方式使得宏比起编译器来说，更像一个函数。

数，因此允许使用局部变量。

第 16 行：不要弄错，这只是局部变量的名字。

第 17 行：这就是内联汇编程序，编译时不需要优化。

第 23 行：参数 1 被用作返回地址。

第 17~24 行：\n\t 用在编译程序/汇编程序接口中，每条汇编指令各占一行。

第 26 行：prev->thread.esp 和 prev->thread.eip 都是输出参数。

[%0]= (prev->thread.esp) 是只写的内存单元；

[%1]= (prev->thread.eip) 是只写的内存单元。

第 27 行：[%2]= (last)，只写寄存器 eax。

[%3]= (esi)，只写寄存器 esi；

[%4]= (edi)，只写寄存器 edi。

第 28 行：这是输入参数。

[%5]= (next->thread.esp)，内存单元。

[%6]= (next->thread.eip)，内存单元。

第 29 行：[%7]= (prev)，重新使用 2 号参数（寄存器 eax）作为输入。

[%8]= (next)，赋给寄存器 edx 的一个输入。

注意：此处没有已修改的寄存器列表。

PowerPC 内联汇编程序与 x86 内联汇编程序的结构几乎完全一样。“m”、“r” 这样的一般约束条件与一组 PowerPC 的平台相关约束条件一起使用。以下是一个交换 32 位指针的例程，注意看看其内联汇编语法与 x86 的语法有多神似：

```
-----
include/asm-ppc/system.h
103 static __inline__ unsigned long
104 xchg_u32(volatile void *p, unsigned long val)
105 {
106     unsigned long prev;
107
108     __asm__ __volatile__ ("\n\
109 1:  lwarx  %0,0,%2 \n\
110
111 " stwcx.  %3,0,%2 \n\
112 bne- 1b"
113 : "=&r" (prev), "=m" (*(volatile unsigned long *)p)
114 : "r" (p), "r" (val), "m" (*(volatile unsigned long *)p)
115 : "cc", "memory");
116
117     return prev;
118 }
-----
```

第 103 行：这一子例程将被原地扩展，而不会被调用。

第 104 行：带参数 `p` 和 `val` 的例程名。

第 106 行：局部变量 `prev`。

第 108 行：内联汇编程序，编译时无须优化。

第 109~111 行：`lwarx` 和 `stwcx` 形成“原子交换操作”。`lwarx` 从内存加载一个字并保存其地址，用于存储其后 `stwcx` 的结果。

第 112 行：不相等时转到标号 1 处（b：向后跳转）。

第 113 行：这是输出操作数。

`[%0] = (prev)`，只写，已修改；

`[%1] = (*(volatile unsigned long *)p)`，只写内存操作数。

第 114 行：这是输入操作数。

`[%2] = (p)`，寄存器操作数；

`[%3] = (val)`，寄存器操作数；

`[%4] = (*(volatile unsigned long *)p)`，存储器操作数；

第 115 行：这是被修改过的操作数。

`[%5] = 状态寄存器被改变；`

`[%6] = 内存单元被修改。`

关于汇编语言以及 Linux 2.6 内核如何应用汇编语言的讨论到此结束。我们先后分析了 PPC 和 x86 体系结构下汇编语言之间的差别，以及与平台无关时如何应用常见的 ASM 程序设计技术。Linux 内核的大部分代码都是用 C 语言编写的，现在，是时候来关注程序设计语言 C 了，当然，还要探讨程序员们在使用 C 语言的过程中遇到的一些常见问题。

2.5 特殊的 C 语言用法

Linux 内核中的许多规范都需要经过反复查找和阅读才能发现其最终的意义和目的。本节主要讨论在整个 Linux 2.6 内核中的常见 C 语言规范，澄清 C 语言用法中几个含糊不清、容易误解的地方。

2.5.1 `asmlinkage`

`asmlinkage` 告诉编译器将参数存入局部栈，这就涉及宏 `FASTCALL`，它通知（与体系结构相关的）编译器将参数传给通用寄存器。下面是 `include/asm/linkage.h` 中的宏代码：

```
-----
include/asm/linkage.h
4 #define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
5 #define FASTCALL(x) x __attribute__((regparm(3)))
6 #define fastcall __attribute__((regparm(3)))
-----
```

下面是 `asmlinkage` 的一个例子：

```
asmlinkage long sys_gettimeofday(struct timeval __user *tv, struct timezone __user *tz)
```

2.5.2 UL

UL 常用在数值常数之后，标明该常数为无符号长整型。UL（或 L）负责告诉编译器将这一数值当做长整型数值来处理，因此，使用 UL（或是 L）很有必要，它能够保证特定体系结构内的数据不会溢出其数据类型所规定的范围。例如：一个 16 位整数的数值范围为 $[-32768, +32767]$ ，一个无符号整数的值最大可以达到 65 535。涉及很大的数或长的位掩码时，使用 UL 有助于编写出与体系结构无关的代码。

内核代码中有一些这样的例子，例如：

```
-----
include/linux/hash.h
18 #define GOLDEN_RATIO_PRIME 0x9e370001UL
-----
include/linux/kernel.h
23 #define ULONG_MAX (~0UL)
-----
include/linux/slab.h
39 #define SLAB_POISON 0x00000800UL /* Poison objects */
-----
```

2.5.3 内联

关键字 **inline** 表明要优化函数的可执行代码，这可以通过将函数的代码合并到调用程序的代码中来实现。Linux 内核使用的内联函数大多被声明为 **static** 类型。一个声明为“**static inline**”的函数促使编译器尝试着将其代码合并到所有调用它的程序中。可能的话，丢弃该函数的汇编代码。有时候，编译器也不能丢弃这些汇编代码（例如，递归函数中的汇编代码就不能被丢弃），但就绝大多数情况而言，声明为静态内联的函数意味着直接将它加入到调用程序中。

这一合并能够消除函数调用的任何开销，**#define** 语句也可以消除函数调用开销，其典型应用参见嵌入式系统交叉编译器的可移植性。

既然如此，为何不始终使用内联呢？原因在于使用内联会增加二进制映像的大小，而这可能会降低访问 CPU 高速缓存的速度。

2.5.4 const 和 volatile

对许多新手而言，这两个关键字让他们很困惑。**const** 不一定代表常数，有时它表示只读的意思。例如，**const int *x** 中的 **x** 是一个指向 **const** 整数的指针，因此，可以修改该指针，但不能修改这个整数，而在 **int const *x** 中，**x** 却是一个指向整数的 **const** 指针，因而这个整数可以改变，但指针 **x** 却不行。以下是一个有关 **const** 的例子：

```
-----
include/asm-i386/processor.h
628 static inline void prefetch(const void *x)
629 {
630     __asm__ __volatile__ ("dcbt 0,%0" : : "r" (x));
631 }
-----
```

关键字 `volatile` 表明变量无需警告就可以被修改。它通知编译器每次使用这个标记的变量时都要重新加载其值，而不是存储和访问一个副本。中断处理、硬件寄存器以及并发进程之间共享的变量都是典型的被标记为 `volatile` 的例子。以下是一个如何使用 `volatile` 的例子：

```
-----
include/linux/spinlock.h
51 typedef struct {
...
volatile unsigned int lock;
...
58 } spinlock_t;
-----
```

假定 `const` 是“只读”的意思，那么，某些特定的变量就可以既是 `const` 类型又是 `volatile` 类型（比如，一个保存某一只读硬件寄存器内容、其值有规律地改变的变量）。

该 C 语言用法掠影将引导未来的 Linux 内核设计者们，在阅读内核源代码时回到正确的轨道上来。

2.6 内核探索工具一览

成功编译并构建自己的 Linux 内核后，你也许很想窥视内核在运行前、运行后，甚至运行过程中其内部的奥秘。本节大致介绍 Linux 内核中常用的分析各种内核文件的工具。

2.6.1 `objdump/readelf`

`objdump` 和 `readelf` 实用程序可分别用于显示目标文件（对 `objdump` 而言）和 ELF 文件（对 `readelf` 而言）中的任何信息。我们可以借助于命令行参数使用命令来查看给定目标文件的文件头、文件大小及结构。例如，以下是一个简单的 C 程序（`a.out`）的 ELF 文件头，它所用的 `readelf` 标志为 `h`：

```
Lwp> readelf -h a.out
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:          ELF32
  Data:           2's complement, little endian
  Version:        1 (current)
  OS/ABI:         UNIX - System V
  ABI Version:    0
  Type:           EXEC (Executable file)
  Machine:        Intel 80386
  Version:        0x1
  Entry point address: 0x8048310
  Start of program headers: 52 (bytes into file)
  Start of section headers: 10596 (bytes into file)
  Flags:          0x0
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 6
  Size of section headers: 40 (bytes)
```

```
Number of section headers: 29
Section header string table index: 26
```

这是该程序使用 `readelf` 标志 `-l` 时的文件头：

```
lwp> readelf -l a.out
Elf file type is EXEC (Executable file)
Entry point 0x8048310
There are 6 program headers, starting at offset 52
Program Headers:
Type   Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
PHDR   0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
INTERP 0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R 0x1
  [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD   0x000000 0x08048000 0x08048000 0x00498 0x00498 R E 0x1000
LOAD   0x000498 0x08049498 0x08049498 0x00108 0x00120 RW 0x1000
DYNAMIC 0x0004ac 0x080494ac 0x080494ac 0x000c8 0x000c8 RW 0x4
NOTE   0x000108 0x08048108 0x08048108 0x00020 0x00020 R 0x4
Section to Segment mapping:
Segment Sections...
00
01 .interp
02 .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel
.plt .init .plt .text .fini .rodata
03 .data .eh_frame .dynamic .ctors .dtors .got .bss
04 .dynamic
05 .note.ABI-tag
```

2.6.2 hexdump

命令 **hexdump** 可以显示给定十六进制/ASCII 码/八进制格式文件的内容。[注意，在老版本的 Linux 中，也使用 `od` (octal dump)。现在，绝大多数系统用 `hexdump` 取代了 `od`。]

例如，要查看一个 ELF 文件 `a.out` 前 64 字节的十六进制表示的话，可以输入如下命令：

```
lwp> hexdump -x -n 64 a.out

00000000 457f 464c 0101 0001 0000 0000 0000 0000
00000010 0002 0003 0001 0000 8310 0804 0034 0000
00000020 2964 0000 0000 0000 0034 0020 0006 0028
00000030 001d 001a 0006 0000 0034 0000 8034 0804
00000040
```

值得注意的是（字节交换的）ELF 文件头在地址 `0x00000000` 处的魔数。

它在调试时特别有用：当硬件设备将其状态转储到某个文件时，常规的文本编辑器一般将它解释成包含很多控制字符的文件，`hexdump` 命令使得你无需干预编辑器的转换工作就可以看到文件中究竟包含了些什么。编辑器 `hexedit` 能支持直接修改该文件，而不必先将其内容转换成 ASCII 码（或 Unicode 编码）。

2.6.3 nm

实用程序 **nm** 可以列出指定目标文件中的符号，它能够显示符号的值、类型和名字，虽然不如其他实用程序一样有用，但调试库文件时却能大显身手。

2.6.4 objcopy

当你想要复制一个目标文件而忽略或改变其某方面的内容时，可以使用 **objcopy** 命令。**objcopy** 的常见用法是去掉测试后正在运行的目标文件中的调试符号，这样做可以大大减小目标文件的大小，因而常用于嵌入式系统中。

2.6.5 ar

ar（或 **archive**）命令有助于维护链接程序使用的索引函数库。**ar** 命令可以将一个或多个目标文件合并到一个链接库中，也可以从单个链接库中将目标文件分离出来。**ar** 命令最常用于 **Make** 文件，将一些常用的函数合并到单个库文件中。例如，你也许有这样一个例程，它可以分析一个命令文件并提取出特定的数据；或者有这样一个调用，它可以从硬件中特定的寄存器内提取信息。可能有多个可执行程序都要使用这些例程，将它们归档到单个库文件的中心位置，有利于更好地进行版本控制。

2.7 内核发言：倾听来自内核的消息

当你的 Linux 系统运行起来后，内核本身会记录一些消息，并提供整个操作过程中系统的状态信息。本节介绍几种最常见的 Linux 内核向终端用户传递消息的方式。

2.7.1 printk()

最基本的内核消息系统之一是 **printk()** 函数。内核使用 **printk()** 而不是 **printf()** 是因为内核没有链接标准 C 函数库。其实 **printk()** 的接口与 **printf()** 的接口完全一样，它可以在控制台显示多达 1024 个字符。**printk()** 函数工作时首先设法获取控制台信号量，然后将要输出的字符存储到控制台的日志缓冲区，再调用控制台驱动程序来刷新缓冲区。如果 **printk()** 无法获得控制台信号量，就只能把要输出的字符存储到日志缓冲区，并依赖拥有控制台信号量的进程来刷新这个缓冲区。在 **printk()** 存储任何数据到日志缓冲区之前，必须使用日志缓冲区锁，这样才能保证并发调用 **printk()** 时不会相互影响。如果已经获得了控制台信号量，那么刷新日志缓冲区之前，可以多次调用 **printk()**。所以，不能用 **printk()** 语句来标明任何程序的运行时间。

2.7.2 dmesg

Linux 内核有多种方式可以用于存储日志和信息。**sysklogd()** 是 **syslogd()** 和 **klogd()** 的组合（详情可参阅这些命令的相关手册页，此处仅作简单总结）。Linux 内核通过 **klogd()** 发送消息，并给它标上适当的警告级。所有级别的消息都存储在 **/proc/kmsg** 中。**dmesg** 是一个命令行工具，可用于显示存储在 **/proc/kmsg** 中的缓冲区内容，并能够根据消息级别来选择是否要过滤这个缓冲区。

2.7.3 /var/log/messages

Linux 系统的 **/var/log/messages** 下存储的是大多数已记录的系统的信息。对于某些存储了已接收消息的特定位置，可以借助 **syslogd()** 程序来读取 **/etc/syslogd.conf** 的内容。根据 **syslogd.conf**

中项的不同，日志信息可以存储到许多不同的文件中去，但该文件在不同版本的 Linux 系统中有所不同，不过通常是放在 `/var/log/messages` 下。

2.8 其他奥秘

本节将介绍当内核设计者们开始漫步在内核代码中时，那些曾经困扰过他们的问题，旨在鼓励你探索 Linux 内部的奥秘。

2.8.1 `__init`

宏 `__init` 告诉编译器相关的函数或变量仅用于初始化。编译器将标有 `__init` 的所有代码存储到特殊的内存段中，初始化结束后就释放这段内存：

```
-----
drivers/char/random.c
679 static int __init batch_entropy_init(int size, struct entropy_store *r)
-----
```

例如，随机设备驱动程序在被加载前初始化一个熵池，加载该驱动程序时，可以使用不同的函数来扩大或减小熵池。对这种设备驱动的初始化过程一般用 `__init` 标记，如果算不上标准的话，这种做法也算是一个惯例。

类似地，如果某些数据只在初始化时才用到，这些数据就必需用 `__initdata` 标记。我们来看看在 ESP 设备驱动程序中如何使用 `__initdata`：

```
-----
drivers/char/esp.c
107 static char serial_name[] __initdata = "ESP serial driver";
108 static char serial_version[] __initdata = "2.2";
-----
```

同样，宏 `__exit` 和 `__exitdata` 仅用于退出和关闭例程，一般在注销设备驱动程序时才使用。

2.8.2 `likely()` 和 `unlikely()`

Linux 内核开发者用宏 `likely()` 和 `unlikely()` 提示编译器和芯片集。现代 CPU 具有众多的启发式分支预测法，它尝试着预测即将到来的命令，以便达到最快的速度。宏 `likely()` 和 `unlikely()` 允许开发者通过编译器告诉 CPU：某一段代码很可能被执行，因而应该被预测到；某一段代码很可能不被执行，不必预测。

如果对指令流水线技术有一定理解的话，就能明白分支预测的重要性了。现代处理器可以预取指令，就是说，它们预测接下来会被执行的几条指令，并把这些指令加载到处理器中，检测后根据执行它们的最佳方式将它们分派到处理器的各种单元中（例如整数、浮点数等）。处理器可能会推迟某些指令的执行，以便等待前一指令执行后产生的中间结果。现在，想象有一个指令流，处理器加载了其中一个分支指令，使得它有两个继续预取指令的指令流，若处理器常常无法做出正确的选择，它将花费很多时间来重新加载需要执行的指令的管道。处理器如何得到分支程序可能会执行的分支的暗示呢？在某些体系结构中，分支预测的一种简单方法就是考查这个分支的目标地址，若其值在当前地址之前，那么，这一分支很可能处于一个循环结构的末端，它循环执行

多次，只有一次失败，这时就会退出循环。

借助于特殊的助记符，可以利用软件来克服体系结构中的相关分支预测。编译器通过函数 `__builtin_expect()` 来实现这一功能，而 `__builtin_expect()` 正是实现宏 `likely()` 和 `unlikely()` 的基础。

如上所述，分支预测和处理器流水线技术都是错综复杂的，也不在本书的讨论之列。但是，“调优”那些我们认为很重要的代码总可以提高性能。现在，来考虑以下代码块：

```
-----
kernel/time.c
90 asmlinkage long sys_gettimeofday(struct timeval __user *tv, struct timezone __user *tz)
91 {
92     if (likely(tv != NULL)) {
93         struct timeval ktv;
94         do_gettimeofday(&ktv);
95         if (copy_to_user(tv, &ktv, sizeof(ktv)))
96             return -EFAULT;
97     }
98     if (unlikely(tz != NULL)) {
99         if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
100             return -EFAULT;
101     }
102     return 0;
103 }
-----
```

在这些代码中，我们发现用于获得时间的系统调用可能有一个非空的 `timeval` 结构（见第 92~96 行）。如果它为空，我们就无法填入所请求的时间。时区非空是不可能的（见第 98~100 行）。换言之，调用者通常会查询时间而往往极少询问时区。

`likely()` 和 `unlikely()` 的特殊实现定义如下^①：

```
-----
include/linux/compiler.h
45 #define likely(x) __builtin_expect(!!(x), 1)
46 #define unlikely(x) __builtin_expect(!!(x), 0)
-----
```

2.8.3 IS_ERR 和 PTR_ERR

宏 `IS_ERR` 将负的错误号编码成指针，而宏 `PTR_ERR` 则将该指针恢复成错误号。这两个宏均在 `include/linux/err.h` 中定义。

2.8.4 通告程序链

通告程序链（notifier chain）机制为内核提供它感兴趣的可变异步事件的相关信息。这个通用接口将其可用性扩展到了内核的所有子系统和组件中。

通告程序链（notifier chain）就是一个 `notifier_block` 对象的单链表：

^① 代码引用中的 `__builtin_expect()`，在 GCC 2.96 之前一直是无效的，因为那些版本的 GCC 无法影响分支预测。

```

-----
include/linux/notifier.h
14 struct notifier_block
15 {
16 int(*notifier_call)(struct notifier_block *self, unsigned long, void *);
17 struct notifier_block *next;
18 int priority;
19 };
-----

```

`notifier_block` 包含指向函数 `notifier_call` 的指针，当事件发生时调用该函数，其参数包括指向保存信息的 `notifier_block` 对象的指针、相应事件代码或标志的值，以及指向特定于子系统的数据类型的指针。

`notifier_block` 结构还包含指向链中下一个 `notifier_block` 的指针和优先级声明。

例程 `notifier_block_register()` 和 `notifier_block_unregister()` 分别用于向特定通告程序链注册或注销一个 `notifier_block` 对象。

2.9 小结

本章阐述了许多探索 Linux 内核之前应当了解的背景知识，介绍了两种动态存储方法——链表和二叉搜索树，对这些数据结构的基本了解有助于读者理解进程和分页机制。接下来介绍了汇编语言的基础知识，以便在机器层进行探索和调试。同时，关于内联汇编程序，我们展示了同一函数中同时出现 C 语言代码和汇编代码的情况。最后，本章讨论了研究内核的各个方面所必需的各种命令和函数。

2.9.1 项目：Hellomod

本节介绍了一些基本概念，为稍后介绍的其他 Linux 相关概念和数据结构打下基础。该项目重点在于使用新的 2.6 驱动程序结构来创建一个可加载模块，并为随后的项目编译该模块。设备驱动程序非常复杂，因此，我们只介绍 Linux 模块的基本结构，在后面的项目中再来介绍这个驱动程序。该模块在 PPC 上和 x86 上都可以运行。

2.9.2 第一步：构造 Linux 模块的框架

我们写的第一个模块是基本的“hello world”字符设备驱动程序。首先，考虑该模块的基本代码，然后示范怎样使用新的 2.6 Makefile 系统（详情参见第 9 章）进行编译，最后，分别使用 `insmod` 命令和 `rmmmod` 命令来附加或移除^①该模块。

```

-----
hellomod.c
001
// hello world driver for Linux 2.6

004 #include <linux/module.h>

```

① 要确保在配置内核时允许卸载模块。

```

005 #include <linux/kernel.h>
006 #include <linux/init.h>
007 #MODULE_LICENSE("GPL"); //get rid of taint message

009 static int __init lkp_init( void )
{
    printk("<1>Hello,World! from the kernel space...\n");
    return 0;
013 }

015 static void __exit lkp_cleanup( void )
{
    printk("<1>Goodbye, World! leaving kernel space...\n");
018 }

020 module_init(lkp_init);
021 module_exit(lkp_cleanup);
-----

```

第4行：所有模块都要使用头文件 `module.h`，此文件必须被包含进来。

第5行：头文件 `kernel.h` 包含了常用的内核函数。

第6行：头文件 `init.h` 包含了宏 `__init` 和 `__exit`，它们允许释放内核占用的内存。建议浏览一下该文件中的代码和注释。

第7行：提示可能没有 GNU 公共许可证。有几个宏是在 2.4 版的内核中才开发的（详情参见 `modules.h`）。

第9~12行：这是模块的初始化函数，它必需包含要编译的代码、初始化数据结构等内容。第11行用函数 `printk()` 从内核发送消息，并提示加载模块后从何处读取该消息。

第15~18行：这是模块的退出和清理函数。此处可以放置终止该驱动程序时，所有执行相关清理工作的代码。

第20行：这是驱动程序初始化的入口点。对于内置模块，内核在引导时调用该入口点；对于可加载模块则在该模块插入内核时才调用。

第21行：对于可加载模块，内核在此处调用 `cleanup_module()` 函数，而对于内置的模块，它没有什么作用。

在该驱动程序中，仅有一个初始化点（函数 `module_init`）和一个清理点（函数 `module_exit`）。加载或卸载模块时，内核会来寻找这些函数。

2.9.3 第二步：编译模块

如果你习惯使用老办法来编译内核模块（例如，从 `#define MODULE` 开始的模块），就会发现新的方法有很大变化。即使是首次编译 2.6 的模块，看起来也相当简单。该模块的 `Makefile` 文件基本内容如下：

`Makefile`

```
002 # Makefile for Linux Kernel Primer module skeleton (2.6.7)
```



```
006 obj-m += hellomod.o
```

要注意的是，需要向编译系统特别声明该模块要编译成可加载模块。该 Makefile 文件的命令行调用打包在名为 `doit` 的 `bash` 脚本文件中，如下所示：

```
-----doit
001 make -C /usr/src/linux-2.6.7 SUBDIRS=$PWD modules
-----
```

第 1 行：-C 选项告诉 `make` 程序读取 Makefiles 或做其他任何事之前，先要改变 Linux 源目录（本例中是 `/usr/src/linux-2.6.7`）。

执行 `./doit` 后可得到与以下内容类似的输出结果：

```
Lkp# ./doit
make: Entering directory '/usr/src/linux-2.6.7'
  CC [M]  /mysource/hellomod.o
  Building modules, stage 2
  MODPOST
  CC  /mysource/hellomod.o
  LD [M]  /mysource/hellomod.ko
make: Leaving directory '/usr/src/linux-2.6.7'
lkp# _
```

如果在 Linux 早期的版本上编译过或创建过 Linux 模块，那么此处还有一个链接步骤 `LD`，其输出模块是 `hellomod.ko`。

2.9.4 第三步：运行代码

现在我们已经准备好，可以将新的模块插入到内核中了。可以用命令 `insmod` 来实现，如下所示：

```
lkp# insmod hellomod.ko
```

`lsmod` 命令可以用于检查模块是否被正确插入到内核中了：

```
lkp# lsmod
Module      Size  Used by
hellomod    2696   0
lkp#
```

模块的输出由函数 `printk()` 来产生。该函数默认会输出信息到系统文件 `/var/log/messages` 中。要快速浏览这些消息可以输入如下命令：

```
lkp# tail /var/log/messages
```

这一命令打印日志文件的最后 10 行内容，可以看到我们的初始化信息：

```
...
...
Mar  6 10:35:55 lkp1 kernel: Hello,World! from the kernel space...
```

只要使用 `rmmmod` 命令，后面加上我们在 `insmod` 命令中看到的模块名，就可以从内核中移除

该模块（还可以看到退出时显示的信息）。如下所示：

```
lkp# rmmod hellomod
```

同样，输出的内容也在日志文件中，如下所示：

```
...  
...  
Mar  6 12:00:05 lkp1 kernel: Hello,World! from the kernel space...
```

根据 X 系统的配置或者是否有基本命令行，`printk` 的输出可以在控制台上显示，也可以存放在日志文件中。在下一个项目中，考虑系统的任务变量时会再提到这个问题。

2.10 习题

1. 描述散列表在 Linux 内核中如何实现。
2. 双向链表中的元素有一个 `list_head` 结构，内核采用该结构之前，它已有指向其他相似结构的 `prev` 指针和 `next` 指针。创建一个仅有 `prev` 和 `next` 指针的结构，其目的是什么？
3. 什么是内联汇编？为什么要使用它？
4. 假定要写一个访问串口寄存器的设备驱动程序。你会将这些地址标记为 `volatile` 吗？为什么？
5. 想象一下 `__init` 做些什么。你认为什么类型的函数会使用这个宏？



进程：程序执行的基本模型

本章内容

- 程序
- 进程描述符
- 进程的创建：系统调用 `fork()`、`vfork()` 和 `clone()`
- 进程的生命周期
- 进程的终止
- 了解进程的动态：调度程序的基本构架
- 等待队列
- 异步执行流程

进程 (process) 是程序执行的基本单位，是了解操作系统工作原理的重要概念。非常有必要理解进程和程序之间的差异。程序指的是由若干函数组成的可执行文件，而进程指的是特定程序的一个实例。进程是对硬件所提供的资源进行操作的基本单元，也是顺序执行其实例化程序的基本单位。操作系统根据进程的需要管理和使用系统资源。

计算机能做很多事情，而这最终都是由进程来实现的。进程能够执行的任务包罗万象，上至用户命令的执行，下至系统资源的管理和硬件的访问。在一定程度上，进程是由它要执行的一组指令、寄存器的内容和程序执行时程序计数器以及它们的状态定义的。

进程与任何动态实体类似，会经历各种状态。事实上，每个进程都有一个生命周期：进程被创建后，将存活一段不定长的时间，在此期间，它将经历各种状态的改变，直到最后消亡。图 3-1 大体上显示了进程的生命周期。

当 Linux 系统加电时，它所需要的进程数是未知的。操作系统在运行的过程中将按照实际需要来创建或撤销进程。

进程是由当前已有的进程调用 `fork()` 来创建的。分叉的进程叫做**子进程** (child process)，创建者叫做**父进程** (parent process)。子进程和父进程继续并发运行。如果父进程继续繁衍更多的子进程，那么新创建的这些进程就是第一个子进程的**兄弟进程** (sibling process)。子进程们还可以依次繁衍出它们自己的子进程。这样就建立起定义关系的进程之间的一种层次关系。

进程被创建后，就准备变为**运行进程** (running process)。这就意味着内核已经为 CPU 执行进程建立起所有的结构，并获取所有必要的信息。当一个进程准备变为运行进程但还没有被选中运

行时，它处于就绪状态。当它变为运行进程后，就能够：

- 被“取消”，被调度程序置为就绪状态(ready)；
- 被中断，并转入等待或阻塞(blocked)状态；
- 变为僵死(zombie)状态，踏上进程消亡之路。这可以通过调用 `exit()` 来实现。

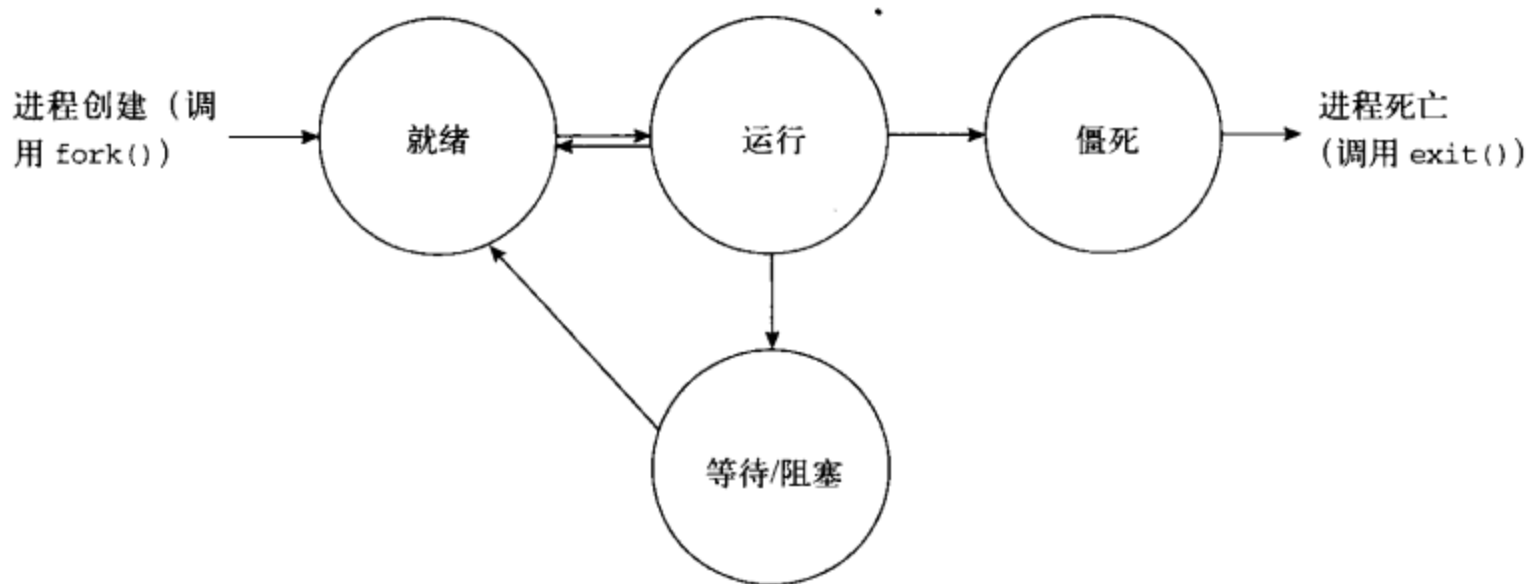


图 3-1 进程的生命周期

本章将详细考察所有这些状态及状态间的转换。调度程序负责选中或取消 CPU 将要执行的进程，我们在第 7 章再详细地讨论调度程序。

程序包含位于内存的多个组成部分，执行程序的进程将根据需要来访问这些内容，包括文本段(text segment)、数据段(data segments)、栈(stack)和堆(heap)。文本段存放 CPU 所执行的指令；数据段存放进程操作的所有数据变量；栈存放自动变量和函数数据；堆存放动态内存分配的数据。当进程被创建时，子进程收到父进程的数据副本，包括数据空间、堆、栈和进程描述符。下一节将详细介绍 Linux 的进程描述符。

人们可以采用多种方式来解释进程，而本书是从进程执行的高层面开始，然后追踪到内核层面，其间穿插说明内核对它所给予的支持。

作为程序员，我们熟悉编写、编译和执行程序。但是，如何把这些与进程联系起来呢？本章将讨论一个示例程序，并从该程序的创建开始，追踪其某些关键任务的执行。在本例中，Bash shell 进程创建的进程将实例化该示例程序；接下来，再由示例程序实例化另外一个子进程。

在继续讨论进程之前，需要解释几个命名约定。通常，术语进程和任务指的是同一件事。当说到运行进程时，就是指 CPU 当前正在执行的进程。

用户态与内核态的比较

我们说一个程序正在用户态(User Mode)下运行或正在内核态(Kernel Mode)下运行，这到底意味着什么呢？在进程的生命周期中，它或者执行自己的代码，或者执行内核代码。所谓内核代码是指当系统调用被执行时、异常发生时或者中断到来时(在中断处理程序中)执行的代码。进程使用的代码如果不是系统调用，就是用户态的代码，因此，进程正运行在用户态，

也服从处理器对它所施加的限制。如果进程正在执行系统调用中，我们说它正运行在内核态。从硬件的角度来看，Intel 处理器上的内核代码就是运行在 0 环上，PowerPC 上的内核代码就是运行在超级用户模式下。

3.1 程序

本节引入一个简单的示例程序，叫做 `create_process`。这个 C 程序阐明了进程可能经历的各种状态、系统调用（这将会导致进程在不同状态间转换）以及对支持进程执行的内核对象的操作，其目的是帮助大家理解程序是如何被实例化成进程的，操作系统又是如何处理进程的。

```
-----
create_process.c
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5
6  int main(int argc, char *argv[])
7  {
8      int fd;
9      int pid;
10
11
12      pid = fork();
13      if (pid == 0)
14      {
15          execl("/bin/ls", NULL);
16          exit(2);
17      }
18
19      if(waitpid(pid) < 0)
20          printf("wait error\n");
21
22      pid = fork();
23      if (pid == 0){
24          fd=open("Chapter_03.txt", O_RDONLY);
25          close(fd);
26      }
27
28      if(waitpid(pid)<0)
29          printf("wait error\n");
30
31
32      exit(0);
33  }
-----
```

这个程序定义了一个**执行上下文**（context of execution），它包括满足程序定义的要求所需资源的相关信息。例如，在任一时刻，CPU 只严格执行一条从内存取得的指令^①。但是，如果没有围绕这条指令的上下文，就无从得知所引用的指令与程序逻辑之间究竟有着怎样的关系，那么，这条指令也就没有任何意义。进程的上下文由程序计数器、寄存器、内存和文件（或者所访问的

① 回想一下以前提到的代码段。

硬件)中存储的值组成。

这个程序编译并链接后产生一个可执行文件,其中存放着执行这个程序所需要的全部信息。本书第9章详细讨论了程序地址空间的划分,以及当讨论进程映像和二进制格式时,地址空间的划分和程序所引用的信息有怎样的关系。

进程具有很多特征,这些特征不仅描述了进程本身,还使它有别于其他进程。进程管理所必需的特征存放在一个单独的数据类型中,那就是所谓的**进程描述符**。在深入研究进程管理之前,有必要讨论一下进程描述符。

3.2 进程描述符

在内核中,进程描述符是一个名为 `task_struct` 的结构体,用于存放进程的属性和信息,与进程相关的所有内核信息都存储在这个结构体中。在其生命周期内,进程要与内核的很多方面——如内存管理和进程调度等打交道,因此,进程描述符除了记录 UNIX 进程的标准属性外,还要记录上述交互过程的相关信息。内核采用循环双向链表 `task_list` 来存放所有进程描述符,并借助全局变量 `current` 来存放当前运行进程的 `task_struct` 的引用。(本书中提及 `current` 时,指的就是当前运行进程的进程描述符。)

进程可能由一个或多个线程组成,每个线程都对应一个 `task_struct`,其中包括一个唯一的线程 ID。在一般的进程中,线程共享相同的内存地址空间。

以下是在进程的生命周期中,进程描述符必须记录的信息:

- ☐ 进程的属性;
- ☐ 进程间的关系;
- ☐ 进程的内存空间;
- ☐ 文件管理;
- ☐ 信号管理;
- ☐ 进程的信任状;
- ☐ 资源限制;
- ☐ 与调度相关的字段。

接下来详细讨论 `task_struct` 结构体中的各个字段。本节将描述这些字段的作用以及与之相关的实际处理过程。虽然许多字段用于描述与上述各类信息相关的活动,但某些内容已经超出本书的范围。`task_struct` 结构的定义位于 `include/linux/sched.h` 中。

```
-----
include/linux/sched.h
384  struct task_struct {
385      volatile long state;
386      struct thread_info *thread_info;
387      atomic_t usage;
388      unsigned long flags;
389      unsigned long ptrace;
390
391      int lock_depth;
```

```
392
393     int prio, static_prio;
394     struct list_head run_list;
395     prio_array_t *array;
396
397     unsigned long sleep_avg;
398     long interactive_credit;
399     unsigned long long timestamp;
400     int activated;
401
402     unsigned long policy;
403     cpumask_t cpus_allowed;
404     unsigned int time_slice, first_time_slice;
405
406     struct list_head tasks;
407     struct list_head ptrace_children;
408     struct list_head ptrace_list;
409
410     struct mm_struct *mm, *active_mm;
411     ...
413     struct linux_binfmt *binfmt;
414     int exit_code, exit_signal;
415     int pdeath_signal;
416     ...
419     pid_t pid;
420     pid_t tgid;
421     ...
426     struct task_struct *real_parent;
427     struct task_struct *parent;
428     struct list_head children;
429     struct list_head sibling;
430     struct task_struct *group_leader;
431     ...
433     struct pid_link pids[PIDTYPE_MAX];
434
435     wait_queue_head_t wait_chldexit;
436     struct completion *vfork_done;
437     int __user *set_child_tid;
438     int __user *clear_child_tid;
439
440     unsigned long rt_priority;
441     unsigned long it_real_value, it_prof_value, it_virt_value;
442     unsigned long it_real_incr, it_prof_incr, it_virt_incr;
443     struct timer_list real_timer;
444     unsigned long utime, stime, cutime, cstime;
445     unsigned long nvcs, nivcs, cnvcs, cnivcs;
446     u64 start_time;
447     ...
450     uid_t uid, euid, suid, fsuid;
451     gid_t gid, egid, sgid, fsgid;
452     struct group_info *group_info;
453     kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
454     int keep_capabilities:1;
455     struct user_struct *user;
```

```

...
457     struct rlimit rlim[RLIM_NLIMITS];
458     unsigned short used_math;
459     char comm[16];
...
461     int link_count, total_link_count;
...
467     struct fs_struct *fs;
...
469     struct files_struct *files;
...
509     unsigned long ptrace_message;
510     siginfo_t *last_siginfo;
...
516 };

```

3.2.1 与进程属性相关的字段

进程的属性分类是我们为进程特征定义的包罗万象的分类，这些特征与进程的状态和标识相关。任何时候检查这些域的值均为内核黑客提供了获取进程当前状态的机会。图 3-2 说明了 task_struct 中与进程属性相关的域。

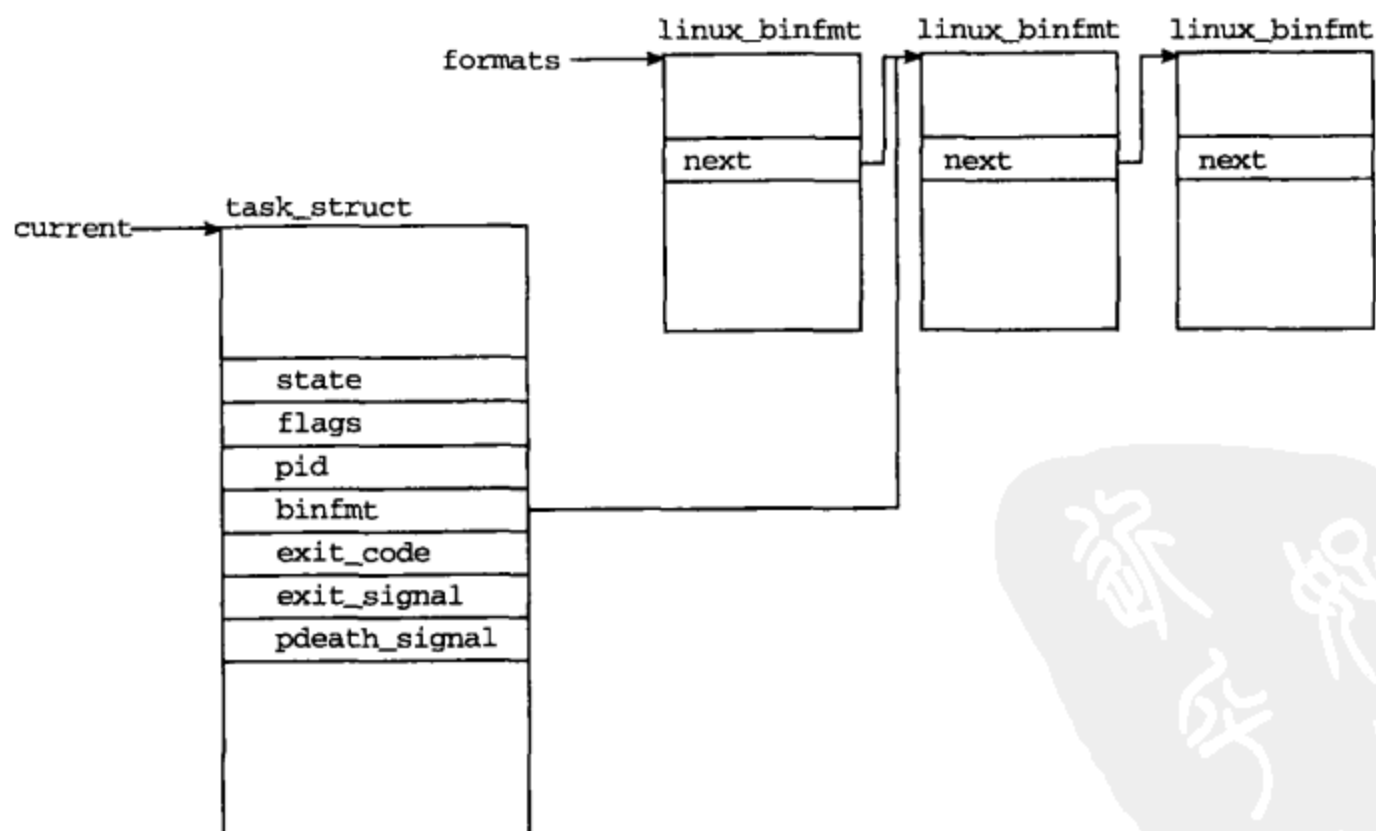


图 3-2 与进程属性相关的字段

1. state

State 字段用于记录进程的状态，在其执行生命周期内，进程可以从该域中找到自己所处的状态，其可能的值有 `TASK_RUNNING`、`TASK_INTERRUPTIBLE`、`TASK_UNINTERRUPTIBLE`、`TASK_ZOMBIE`、`TASK_STOPPED` 以及 `TASK_DEAD`（详见本章 3.4 节）。

2. pid

在 Linux 中,每个进程都有唯一的进程标识符 (process identifier) `pid`。`pid` 位于 `task_struct` 结构体中,类型为 `pid_t`,若追本溯源的话,`pid_t` 是整数类型,但 `pid` 默认的最大值是 32 768 (短整型的最大可能值)。

3. 标志

标志定义的是进程的特殊属性。每个进程的标志都是在 `include/linux/sched.h` 中定义的,表 3-1 给出了这些标志的值,它们为内核黑客提供了进程所处状态的更多相关信息。

表3-1 `task_struct`结构体中标志字段的部分值

Flag名称	何时设置
<code>PF_STARTING</code>	进程创建时设置
<code>PF_EXITING</code>	调用 <code>do_exit()</code> 时设置
<code>PF_DEAD</code>	进程退出、调用 <code>exit_notify()</code> 时设置。此时,进程的状态是 <code>TASK_ZOMBIE</code> 或 <code>TASK_DEAD</code>
<code>PF_FORKNOEXEC</code>	父进程创建子进程时设置该标志

4. binfmt

Linux 支持多种可执行文件格式。可执行文件格式是为指明程序代码如何被载入内存而定义的一种结构。图 3-2 说明了 `task_struct` 与 `linux_binfmt` 结构体之间的关系。其中,`linux_binfmt` 结构体包含了与特定二进制格式相关的所有信息 (详见第 9 章)。

5. exit_code和exit_signal

`exit_code` 与 `exit_signal` 字段分别存放进程的退出值和终止信号 (如果使用了该字段的话)。这是将子进程的退出值传给其父进程的方式。

6. pdeath_signal

`pdeath_signal` 是父进程消亡时发出的信号。

7. comm

通常通过在命令行调用一个可执行程序来创建进程。调用时,`comm` 字段用于存放该可执行程序的名称。

8. ptrace

当进程因进行性能测定而调用系统调用 `ptrace()` 时设置 `ptrace` 字段。`ptrace()` 的标志在文件 `include/linux/ptrace.h` 中定义。

3.2.2 与调度相关的字段

进程运行时宛如拥有自己的虚拟 CPU,事实上是多个进程共享一个 CPU。为了在运行的进程之间进行切换,每个进程均与调度程序密切相关 (详见第 7 章)。

然而,要理解上述的某些字段,势必要理解调度的基本概念。当多个进程都准备好等待运行时,由调度程序决定谁先运行、运行多长时间。调度程序通过给每个进程分配一个时间片和优先级来实现公平、高效的调度。时间片定义了一个进程被切换掉,运行另一个进程之前允许运行的

时间长度。进程的优先级则是一个数值，它定义了相对其他就绪进程而言，进程被允许执行的相对顺序，优先级越高的进程会越早被调度运行。图 3-3 所示的字段存放了调度所需的值。

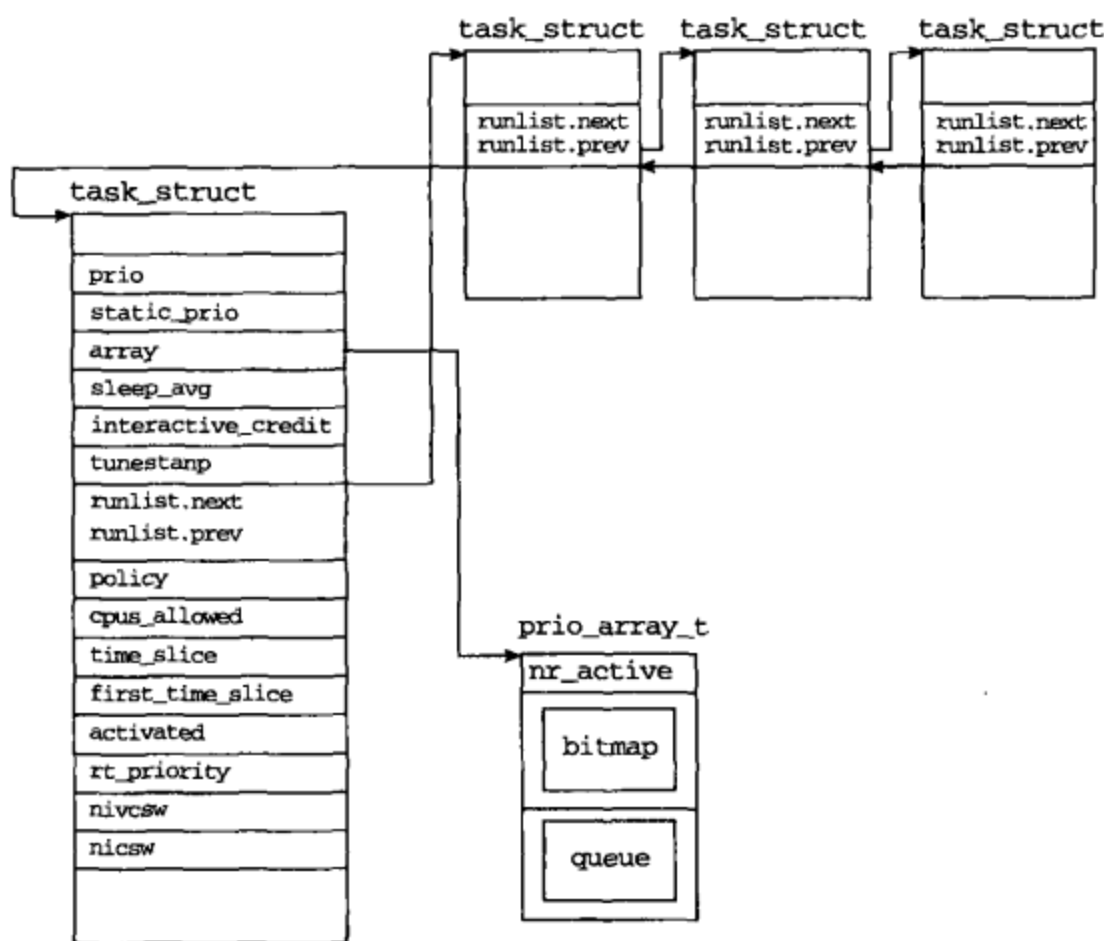


图 3-3 与调度相关的字段

1. prio

在第 7 章，将会看到进程的动态优先级是由进程的调度历史及一个具体的 nice 值确定的值（有关 nice 值的更多说明详见下面的注释框）。当进程时间片用完且进程未被执行时，其动态优先级将在睡眠时被更新，其值 prio 与下面将要说明的 static_prio 字段的值有关。prio 的值是在 static_prio 的基础上加/减 5，具体情况依据进程的历史而定。如果已经睡眠了很长时间的话，那么它将得到 +5 的奖励；反之，如果运行了相当长时间且用完了其时间片，那么它将得到 -5 的惩罚。

2. static_prio

static_prio 的值等于 nice 的值，前者的默认值为 MAX_PRIO-20。在我们的内核中，MAX_PRIO 默认为 140。

nice

系统调用 nice() 允许用户修改进程的静态调度优先级。nice 值的变化范围是 -20~19。函数 nice() 调用 set_user_nice() 来设置 task_struct 中的 static_prio 字段。static_prio 的值根据 nice 值利用宏 PRIO_TO_NICE 来计算。同样，nice 的值是根据 static_prio 值通过调用宏 NICE_TO_PRIO 来计算的。

```
-----kernel/sched.c
#define NICE_TO_PRIO(nice) (MAX_RT_PRIO + nice + 20)
#define PRIO_TO_NICE(prio) ((prio - MAX_RT_PRIO - 20))
```

3. **run_list**

run_list 指向 **runqueue**。**runqueue** 是一个包含所有准备运行的进程的队列，详见 3.6.1 节中的 **runqueue** 结构。

4. **array**

array 指向 **runqueue** 的优先级数组。3.6 节将详细阐述该数组的相关内容。

5. **sleep_avg**

sleep_avg 用于计算进程的有效优先级，它是进程睡眠过程耗费的时钟周期的平均数。

6. **timestamp**

timestamp（时间戳）用于当进程睡眠或放弃处理器时计算 **sleep_avg**。

7. **interactive_credit**

interactive_credit 与 **sleep_avg**、**activated** 共同使用来计算 **sleep_avg** 的值。

8. **policy**

policy 用于确定进程的类型，例如，分时进程和实时进程。进程的类型极大地影响到其调度优先级。有关该字段的内容，详情请参考第 7 章。

9. **cpus_allowed**

cpus_allowed 指明由哪个 CPU 来处理任务。在多处理器系统中，可通过这种方式指定特定的任务在哪个 CPU 上运行。

10. **time_slice**

time_slice 定义进程每次被调度时允许运行的最长时间。

11. **first_time_slice**

first_time_slice 被反复设置为 0 并记录调度时间。

12. **activated**

activated 记录平均睡眠时间的增减。若一个不可中断的睡眠进程被唤醒，该字段将被置为 -1。

13. **rt_priority**

rt_priority 是实时进程的优先级，是一个静态值，只能通过 **schedule()** 来更新，支持实时任务时一定会用到该字段。

14. **nivcsw**和**nvcs**

系统中存在不同类型的上下文切换，内核将记录这些信息，以备分析。全局上下文切换计数器将根据上下文切换中涉及的不同转换种类，被设置成 4 种不同的上下文切换计数器中的一个（详情见 7.1.2 节）。基本上下文切换计数器主要有两种。

- ❑ **nivcsw**（被动的上下文切换次数）字段记录进程被内核抢占的次数，其值仅当进程从内核抢占代码返回时才增加，此时上下文切换计数被置为 **nivcsw**。
- ❑ **nvcs**（主动的上下文切换次数）字段记录除去内核抢占之外的上下文切换次数。若前一

个状态不是主动抢占的话，则上下文切换计数被置为 `nvcs`。

3.2.3 涉及进程间相互关系的字段

结构体 `task_struct` 中的下列字段涉及进程间的相互关系。每个进程 `p` 都有一个创建它的父进程。进程 `p` 本身也可以创建其他进程，因此，它可能有子进程。由于进程 `p` 的父进程可以创建多个进程，进程 `p` 也可能有兄弟进程。图 3-4 给出了这些进程的 `task_struct` 之间有怎样的关系。

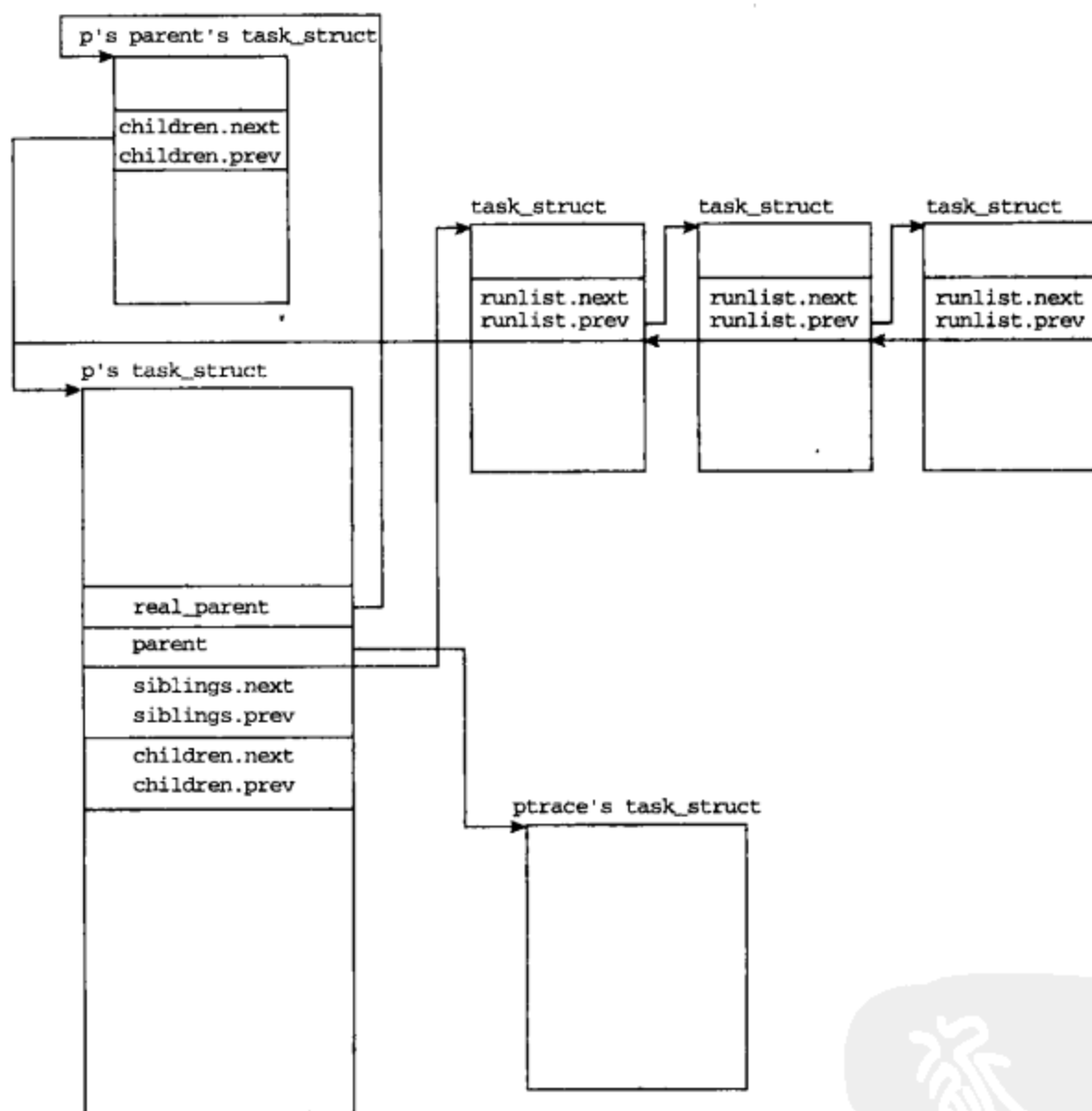


图 3-4 涉及进程间相互关系的字段

1. `real_parent`

`real_parent` 指向当前进程的父进程的进程描述符。若当前进程的父进程已经消亡，那么它指向 `init()` 的进程描述符。在早期的内核版本中，该字段是 `p_opptr`。

2. `parent`

`parent` 是指向其父进程描述符的指针。在图 3-4 中，该指针指向 `ptrace` 的 `task_struct`。当 `ptrace` 在某一进程上运行时，`task_struct` 中的 `parent` 字段指向 `ptrace` 进程。

3. `children`

`children` 是指向当前进程的子进程列表的 `struct`。

4. sibling

sibling 是指向当前进程的兄弟进程列表的 struct。

5. group_leader

进程可以是某一进程组中的成员。每个进程组都有一个被定义为组长的进程。如果进程是某个组的成员，则 group_leader 是一个指向其组长的进程描述符的指针。组长进程通常拥有自己的 tty，即控制终端，进程就是从这个终端创建的。

3.2.4 与进程信任状相关的字段

在多用户系统中，必须能够区分进程究竟是由哪个用户创建的，这一点从系统的安全性和保护用户数据的角度来讲也是必要的。因此，每个进程都有一个信任状，系统根据这个信任状来确定进程可以访问和不可以访问的系统资源。图 3-5 解释了 task_struct 结构体中与进程信任状有关的字段。

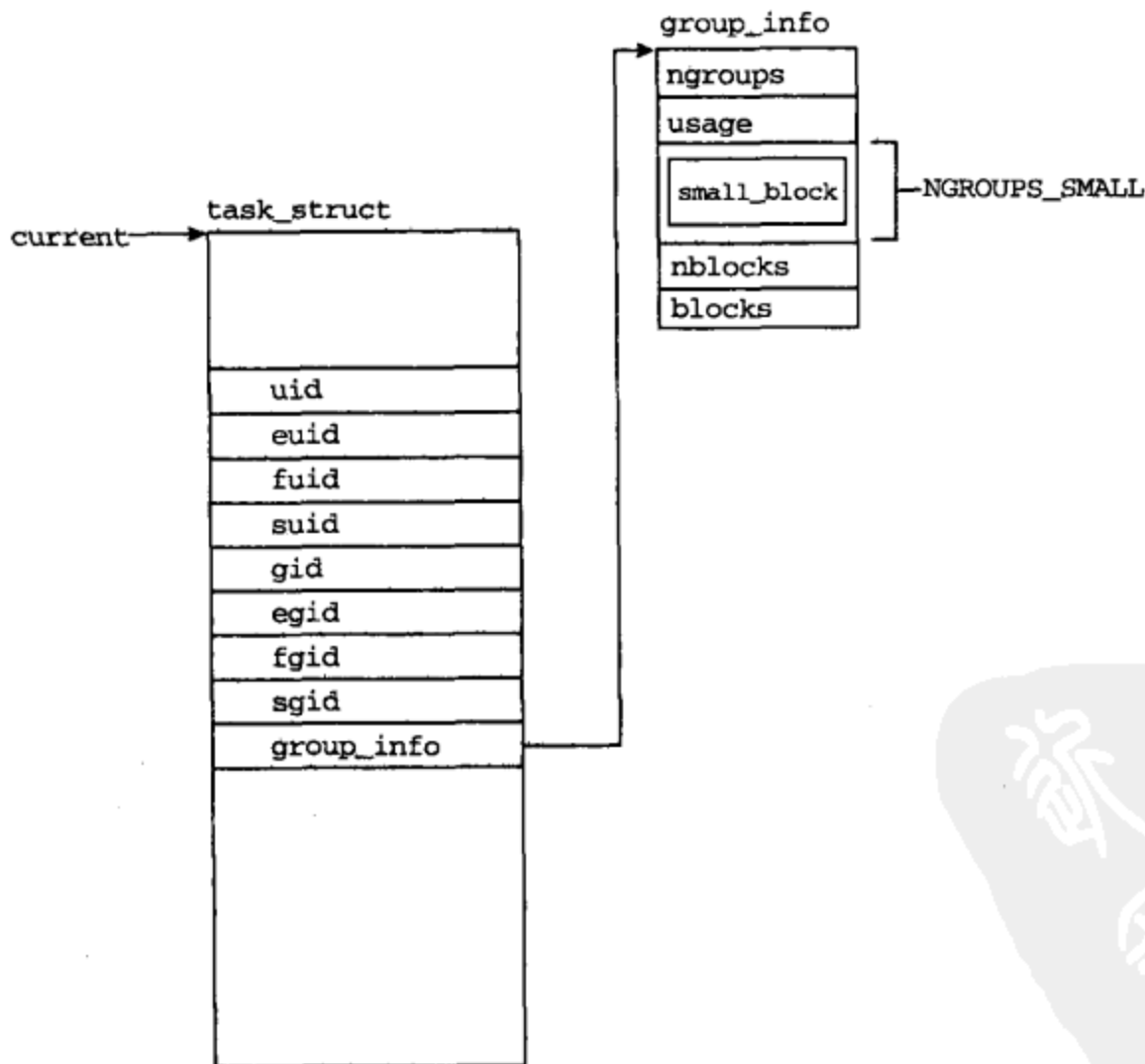


图 3-5 与进程信任状相关的域

1. uid和gid

uid 用于存放创建该进程的用户的 ID，该字段是出于数据保护和系统安全方面的考虑而设置的。同理，gid 是该进程所在进程组的组 ID。进程 0 的 uid 和 gid 分别是 root 的用户 ID 和组 ID。

2. **uid**和**legid**

有效用户 ID 通常存放着和 **uid** 字段相同的值,但如果被执行的程序已将设置 UID 的位(SUID)置位的话,情况就有所不同了。此时,有效用户 ID 指的是程序文件拥有者的 ID。通常,这允许任意用户能够以与另一个用户(比如说, **root**)相同的权限来运行一个特殊的程序。有效组 ID 的作用与之类似,也是只有当设置组 ID 的位(SGID)被置位后,其值才会与 **gid** 不同。

3. **suid**和**sgid**

suid (存放用户 ID) 和 **sgid** (存放组 ID) 用于系统调用 **setuid()**。

4. **fsuid**和**fsgid**

文件系统检测时将专门检查 **fsuid** 和 **fsgid** 的值。它们通常和 **uid**、**gid** 的值相同,除非调用了 **setuid()**。

5. **group_info**

在 Linux 中,一个用户可以是多个组的成员。这些组可能拥有不同的访问系统和存取数据的权限。因此,进程必须继承其信任状。**group_info** 字段是一个指向 **group_info** 类型的结构体的指针,结构体 **group_info** 中存放的是进程所在的不同组的所有信息。

结构体 **group_info** 允许进程在有充足的可用内存时与多个组关联。在图 3-5 中,**group_info** 的字段 **small_block** 就是一个类型为 **gid_t** 的数组,数组中有 **NGROUPS_SMALL** (本例中为 32) 个 **gid_t** 单元。如果某个进程所属的组超过 32 个,内核可以为超出 **NGROUPS_SMALL** 的部分分配块或者页来存放所需的 **gid_t** 数目。**nblocks** 存放已分配的块的数目,而 **ngroups** 存放 **small_block** 数组中单元的个数,该数组的每个单元中存放的都是 **gid_t** 的值。

3.2.5 与进程权能相关的字段

一般来说,UNIX 系统通过为每个给定进程赋予特权级(超级用户或 **UID=0** 的用户)或非特权级(其他任何进程),从而对特定的访问和操作提供与进程相关的保护。在 Linux 中,引入权能来划分以前只有超级用户才能执行的操作,换句话说,权能就是能授予进程的一种或多种特权,这种“授予”彼此相互独立且不依赖于进程的 **UID**。这样一来,特殊的进程可以获得一些权限来执行某些特殊的管理任务,而不必取得所有特权,也无需成为超级用户的进程。因此,权能被定义为给定的管理操作。图 3-6 给出了与进程权能相关的字段。

cap_effective、**cap_inheritable**、**cap_permitted**和**keep_capabilities**

用于支持权能模型的结构是一个 32 位的无符号数,是在文件 **include/linux/security.h** 中定义的。每个 32 位的掩码对应一个权能集,每种权能占用一个二进制位。

□ **cap_effective**: 有效权能。这是当前能够被进程使用的权能。

□ **cap_inheritable**: 可继承权能。这是可以通过系统调用 **execve** 来传递的权能。

□ **cap_permitted**: 许可权能。这可以是有效的权能或者可继承的权能。

要理解上述 3 种类型权能之间的区别,方法之一是考虑与其父进程创建的基因池类似的许可权能。考虑到父进程创建的基因质量,可以只显示其中一个子集(即有效权能),或者传递这个子集(即可继承权能)。许可权能构成了一种可能性,而有效权能把这种可能性变成了现实。

因此, **cap_effective** 和 **cap_inheritable** 永远是 **cap_permitted** 的子集。

□ `keep_capabilities`: 当调用 `setuid()` 时, 记录进程到底是放弃还是保持其权能。表 3-2 列出了一部分在 `include/linux/capability.h` 中定义的权能。

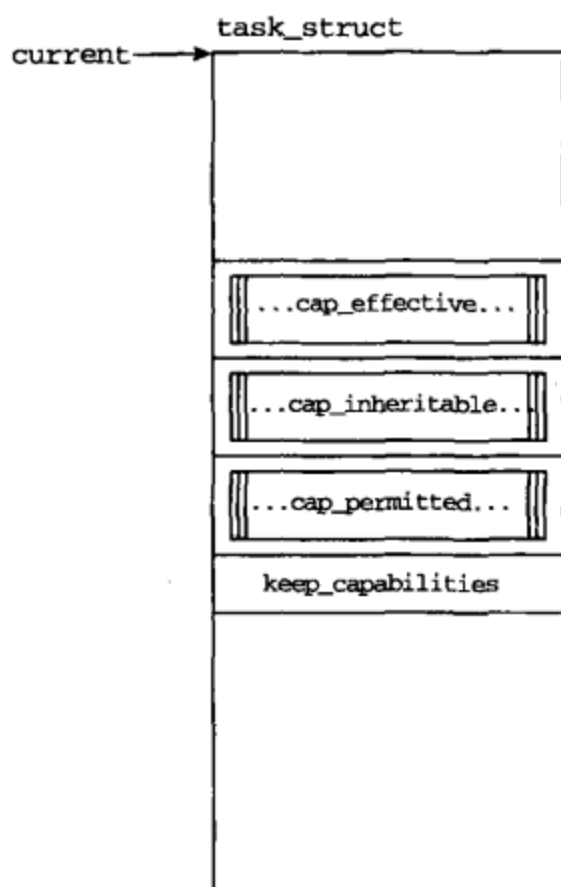


图 3-6 与进程权能相关的字段

表3-2 部分权能

权 能	说 明
<code>CAP_CHOWN</code>	忽略由 <code>chown()</code> 强加的限制
<code>CAP_FOWNER</code>	忽略文件许可的限制
<code>CAP_FSETID</code>	忽略 <code>setuid</code> 和 <code>setgid</code> 对文件施加的限制
<code>CAP_KILL</code>	发送信号时忽略 <code>ruid</code> 和 <code>euid</code>
<code>CAP_SETGID</code>	忽略与组相关的许可验证
<code>CAP_SETUID</code>	忽略与 <code>uid</code> 相关的许可验证
<code>CAP_SETCAP</code>	允许进程设置其权能

通过给 `capable()` 传递权能变量作为参数, 内核调用该函数检查是否设置了特定的权能。通常, 该函数检查在 `cap_effective` 集合中是否设置了该权能的相应位。若已设置, 则将 `current->flags` 置为 `PF_SUPERPRIV`, 意味着授予该权能, 同时返回 1; 否则, 返回 0。

与权能操作有关的系统调用包括 `capget()`、`capset()` 和 `prctl()`。前两个系统调用允许进程获取、设置其权能, 而 `prctl()` 则允许操作 `current->keep_capabilities`。

3.2.6 与进程限制相关的字段

进程通过硬件和调度程序来使用系统中的资源。下列字段可用于记录进程如何使用资源, 以

及该对进程进行何种限制。

rlim

rlim 字段是一个数组，该数组通过维护资源限制值对资源进行控制和记账。图 3-7 解释了 task_struct 中的 rlim 字段。

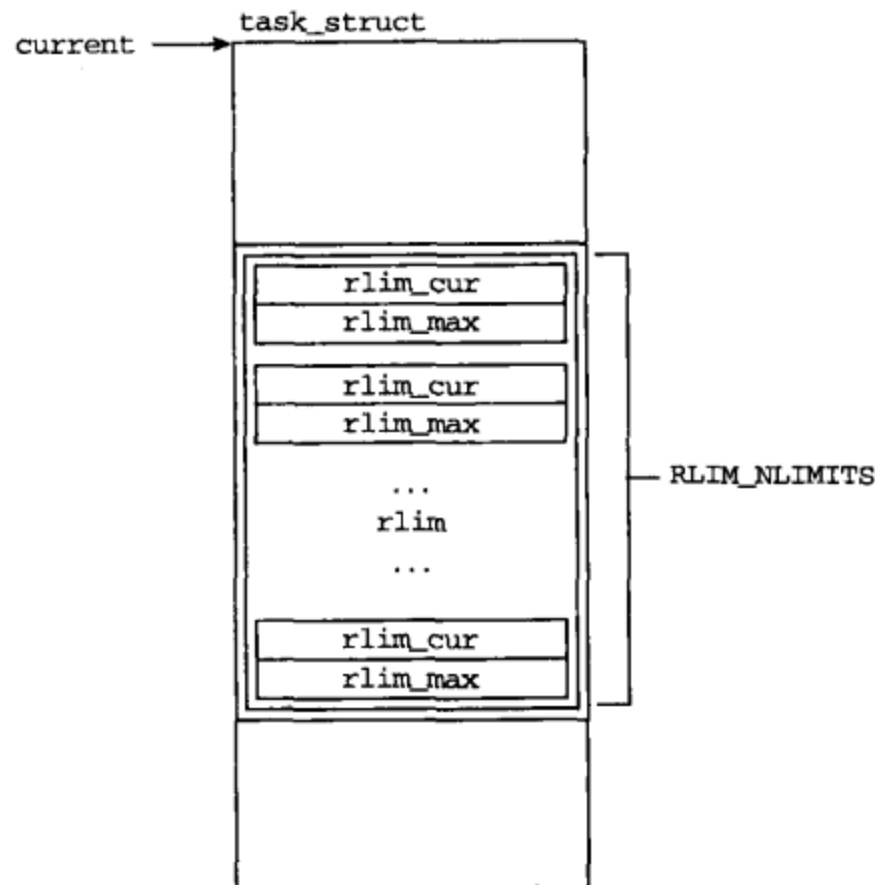


图 3-7 task_struct 中的资源限制

Linux 可识别是否需要限制允许一个进程使用的特定资源的数量。由于不同的进程可能使用的资源种类和数量都不同，因此有必要为每个进程存放这些基本信息。如果我们要保存这些信息，还有什么地方能比进程描述符更为合适呢？

`rlimit` 描述符 (`include/linux/resource.h`) 有 `rlim_cur` 和 `rlim_max` 这两个字段，分别表示资源的当前限制数目以及最大数目限制。这个限制的单位依该结构所指的资源种类的不同而有所不同。

```
-----
include/linux/resource.h
struct rlimit {
    unsigned long    rlim_cur;
    unsigned long    rlim_max;
};
-----
```

表 3-3 列出了一些资源，其限制数目在 `include/asm/resource.h` 中定义。x86 与 PPC 有相同的资源限制列表及其默认值。

表3-3 资源限制的值

RL名称	说 明	默认的rlim_cur	默认的rlim_max
RLIMIT_CPU	该进程使用CPU的时间量，单位为秒	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_FSIZE	文件大小，单位为1 K	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_DATA	堆大小，单位是字节	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_STACK	栈大小，单位是字节	_STK_LIM	RLIM_INFINITY
RLIMIT_CORE	核心转储文件的大小	0	RLIM_INFINITY
RLIMIT_RSS	驻留集（实存）的最大值	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_NPROC	该进程拥有的进程数	0	0
RLIMIT_NOFILE	进程一次最多可打开的文件数	INR_OPEN	INR_OPEN
RLIMIT_MEMLOCK	不可交换（可被锁定）内存的大小	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_AS	进程地址空间的大小，单位是字节	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_LOCKS	文件锁的数目	RLIM_INFINITY	RLIM_INFINITY

当上述的某个值被设置成 RLIM_INFINITY 时，该进程对这个资源的使用不再受限。

当前限制 rlim_cur 是一个软限制，调用 setrlimit() 可以对其进行修改，其最大限制由 rlim_max 来定义，任何没有相应特权的进程都不能超出这个最大值的限制。系统调用 getrlimit() 返回资源限制的值。setrlimit() 和 getrlimit() 都需要以资源的名称和一个指向 rlimit 类型结构的指针作为参数。

3.2.7 与文件系统及地址空间相关的字段

进程总是通过执行打开、关闭、读、写文件这样的任务与文件密切相关。task_struct 结构中有两个字段和“与文件以及文件系统相关”的数据有关：fs 和 files（详细内容请参见第 6 章），和地址空间相关的两个字段是 active_mm 和 mm（详细内容请参见第 4 章），图 3-8 所示为与文件系统和地址空间有关的 task_struct 的字段。

1. fs

fs 字段保存一个指向文件系统信息的指针。

2. files

files 字段保存一个指向进程的文件描述符表的指针。文件描述符也保存一个指针，它指向进程所打开的文件（更确切地说，是指向被打开的文件的描述符）。

3. mm

mm 指向与地址空间及内存管理相关的信息。

4. active_mm

active_mm 是指向最后访问的地址空间的指针。mm 和 active_mm 最初都是指向 mm_struct。

讨论进程描述符能使我们更好地理解进程在其整个生命周期中所涉及的数据及其类型。接下来将讨论在进程的整个生命期内会发生些什么。以下章节将阐释进程的不同生命阶段及其状态，并逐行分析一个实例，以解释在此期间内核中究竟发生了什么。

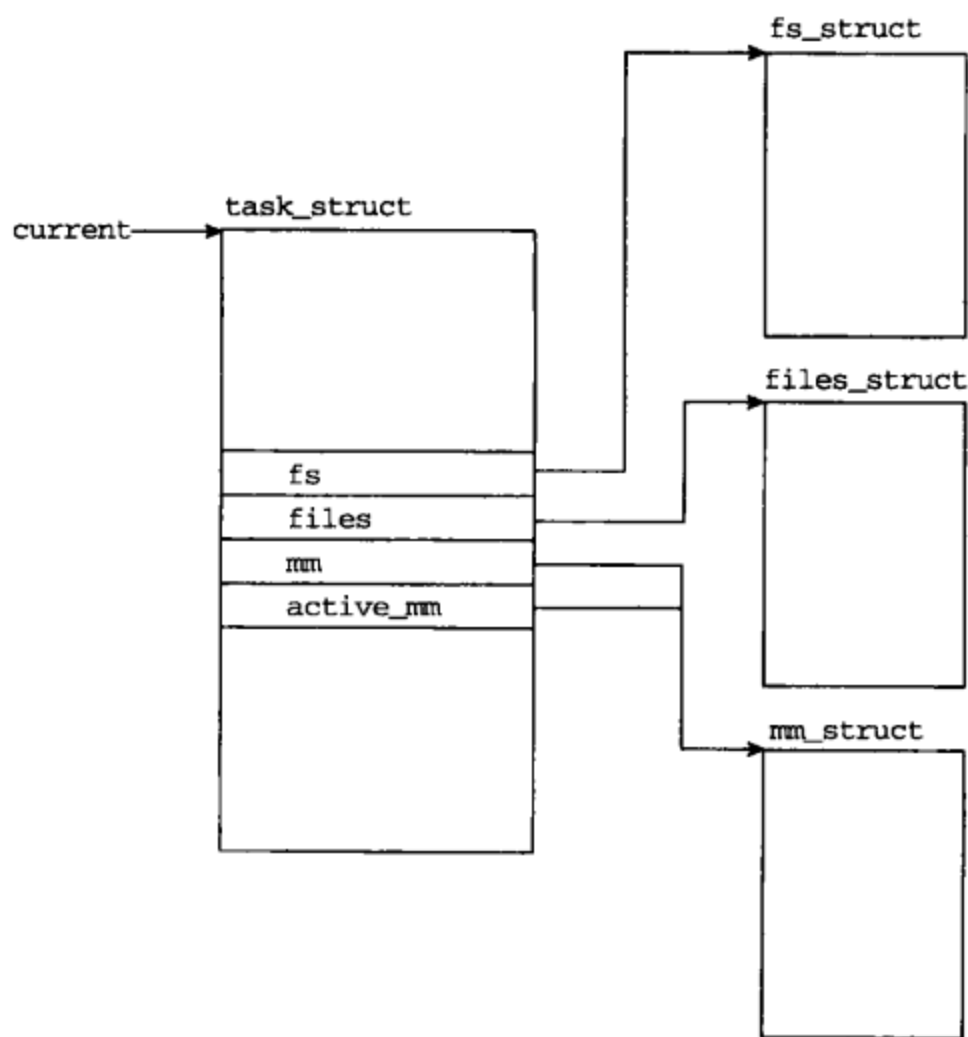


图 3-8 与文件系统和地址空间相关的字段

3.3 进程的创建：系统调用 `fork()`、`vfork` 和 `clone()`

当示例代码被编译成文件后（此处是指 ELF 可执行文件^①），就可以从命令行调用它。看看敲了回车键后会发生什么？前面已经说过，任何进程都是由其他进程创建的。操作系统通过系统调用 `fork()`、`vfork` 和 `clone()` 来完成进程的创建。

C 函数库提供了与这三个系统调用对应的函数。这些函数的原型是在文件 `<unistd.h>` 中声明的。图 3-9 解释了调用 `fork()` 函数的进程如何去执行系统调用 `sys_fork()`。该图描述了内核代码如何执行实际的进程创建。与此类似，`vfork()` 调用 `sys_fork()`，而 `clone()` 调用 `sys_clone()`。

这三个系统调用最终都调用了内核函数 `do_fork()`，该函数完成与进程创建有关的大部分工作。读者也许会奇怪创建进程为什么会有三个不同的函数？这三个函数在如何创建进程上有细微的差别，究竟选择哪一个函数都有其明确的理由。

在 shell 提示符下按下回车键时，shell 将通过调用 `fork()` 来创建新的进程，以便执行我们的程序。实际上，如果在 shell 提示符下输入 `ls` 命令并按回车，shell 在那一刻的伪代码类似于下面的语句：

① ELF 可执行文件是 LINUX 支持的一种可执行文件格式。第 9 章将会讨论这种可执行文件格式。

```

if( (pid = fork()) == 0 )
    execve("foo");
else
    waitpid(pid);

```

现在来分析这几个函数，并且一直向下追踪到其系统调用。尽管我们的程序调用了 `fork()`，但调用 `vfork()` 或 `clone()` 也一样容易，这就是在本节对这三个函数都进行介绍的原因。第一个函数是 `fork()`，它调用了 `sys_fork()`，`sys_fork()` 又调用了 `do_fork()`。然后分析 `vfork()` 和 `clone()`，并跟踪其执行过程，直至调用 `do_fork()`。

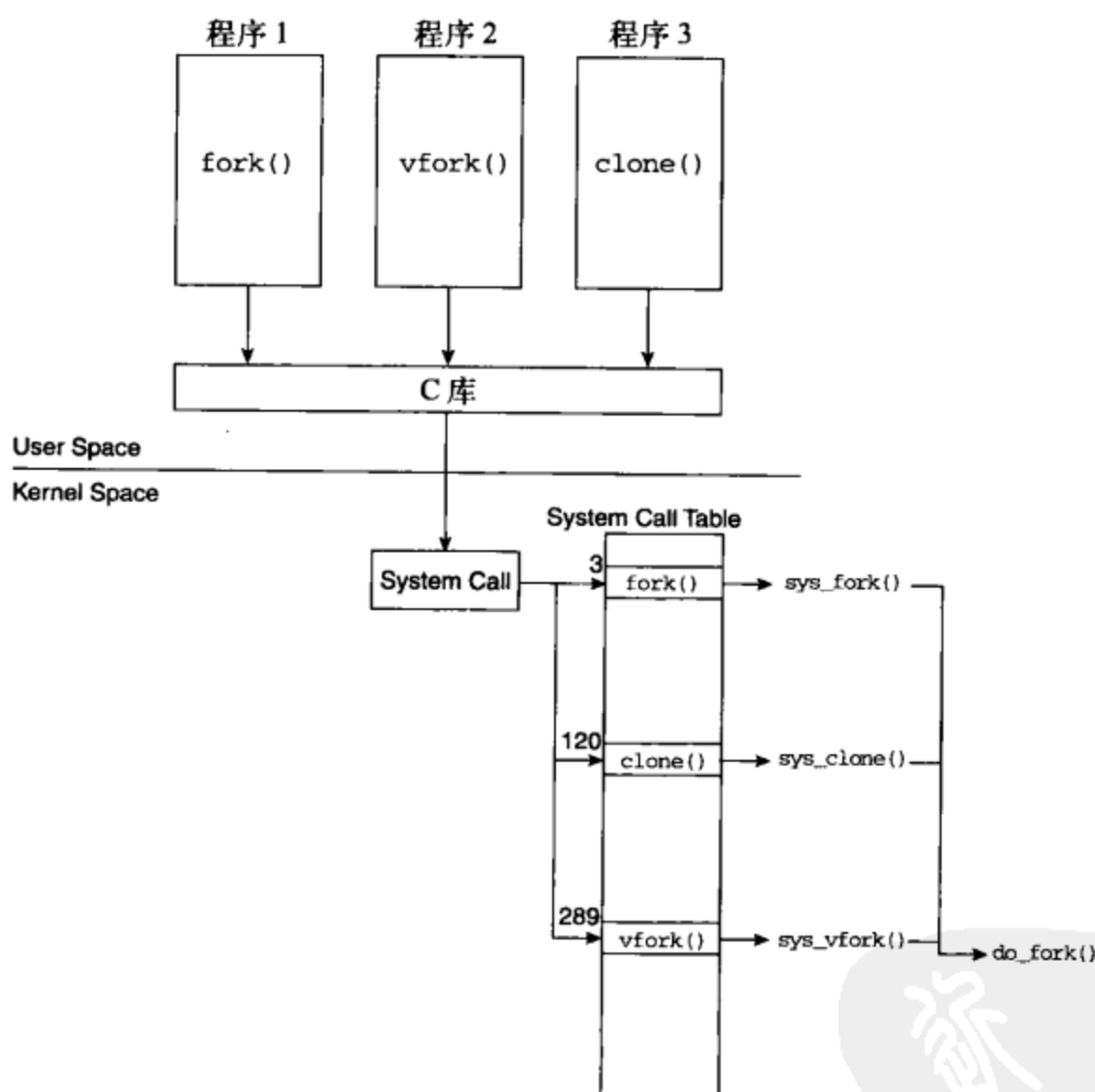


图 3-9 进程创建的系统调用

3.3.1 `fork()` 函数

`fork()` 函数返回了两次：一次是在父进程，一次是在子进程。如果在子进程中返回，将返回 0。如果在父进程中返回，将返回子进程的 PID。当 `fork()` 函数被调用时，该函数把包括该系统调用表索引在内的一些必要信息放入适当的寄存器，系统调用表中存放系统调用的指针由处理器来决定将这些信息放入哪个寄存器中。

关于这个问题，如果读者想继续深入了解，请查看 3.8.2 节中 `sys_fork()` 是如何被调用的。不过，不需理解一个新进程是如何创建的。

现在让我们来看看 `sys_fork()` 函数。这个函数仅仅是调用 `do_fork()` 函数。要注意的是，`sys_fork()` 函数是与体系结构相关的，因为其参数是由系统寄存器传来的。

```
-----
arch/i386/kernel/process.c
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs, 0, NULL, NULL);
}
-----

arch/ppc/kernel/process.c
int sys_fork(int p1, int p2, int p3, int p4, int p5, int p6,
             struct pt_regs *regs)
{
    CHECK_FULL_REGS(regs);
    return do_fork(SIGCHLD, regs->gpr[1], regs, 0, NULL, NULL);
}
-----
```

从上面的代码可以看出，这两种体系结构所接受的系统调用的参数也不同。`pt_regs` 存放了栈指针等信息。实际上，按照惯例，在 PPC 体系结构中，`gpr[1]` 存放了栈指针，而在 x86 体系结构中，`%esp`^① 存放了栈指针。

3.3.2 vfork() 函数

`vfork()` 函数和 `fork()` 函数类似，但 `vfork()` 的父进程一直阻塞，直到子进程调用 `exit()` 或 `exec()` 为止。

```
sys_vfork()
arch/i386/kernel/process.c
asmlinkage int sys_vfork(struct pt_regs regs)
{
    return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.ep, &regs, 0, NULL, NULL);
}
-----

arch/ppc/kernel/process.c
int sys_vfork(int p1, int p2, int p3, int p4, int p5, int p6,
             struct pt_regs *regs)
{
    CHECK_FULL_REGS(regs);
    return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs->gpr[1],
                  regs, 0, NULL, NULL);
}
-----
```

在 `sys_vfork()` 中调用 `sys_fork()` 和直接调用 `sys_fork()` 的唯一区别就在于传给 `do_fork()` 的标志不相同。这些标志随后用来决定是否执行刚才所描述的阻塞的父进程的操作。

① 回忆一下，在“AT&T”格式的代码中，寄存器前要加一个前缀 %。

3.3.3 clone() 函数

库函数 clone() 不同于 fork() 和 vfork(), 它把一个指向函数的指针和该函数的参数作为自己的参数。由 do_fork() 创建的子进程刚一诞生就调用这个函数。

```

-----
sys_clone()
arch/i386/kernel/process.c
asmlinkage int sys_clone(struct pt_regs regs)
{
    unsigned long clone_flags;
    unsigned long newsp;
    int __user *parent_tidptr, *child_tidptr;

    clone_flags = regs.ebx;
    newsp = regs.ecx;
    parent_tidptr = (int __user *)regs.edx;
    child_tidptr = (int __user *)regs.edi;
    if (!newsp)
        newsp = regs.esp;
    return do_fork(clone_flags & ~CLONE_IDLETASK, newsp, &regs, 0,
parent_tidptr, child_tidptr);
}
-----

arch/ppc/kernel/process.c
int sys_clone(unsigned long clone_flags, unsigned long usp,
               int __user *parent_tidp, void __user *child_thread\
ptr,
               int __user *child_tidp, int p6,
               struct pt_regs *regs)
{
    CHECK_FULL_REGS(regs);
    if (usp == 0)
        usp = regs->gpr[1];    /* stack pointer for chi\
ld */
    return do_fork(clone_flags & ~CLONE_IDLETASK, usp, regs,\
0,
                  parent_tidp, child_tidp);
}
-----

```

如表 3-4 所示, fork()、vfork() 和 clone() 的唯一区别在于随后调用 do_fork() 时设置的标志不同。

表3-4 由fork()、vfork()和clone()传递给do_fork的标志

	fork()	vfork()	clone()
SIGCHLD	X	X	
CLONE_VFORK		X	
CLONE_VM		X	

最后，来看看 `do_fork()`，它才真正创建进程。到此为止，父进程调用了 `fork()`，然后它才能调用系统调用 `sys_fork()`；但仍然没有产生新的进程。程序 `foo` 仍然作为可执行文件存放在磁盘上，它还没有运行，也没有装入内存。

3.3.4 `do_fork()` 函数

我们逐行追踪 `do_fork()` 在内核中的执行，描述一下新进程创建背后的细节。

```
-----
kernel/fork.c
1167 long do_fork(unsigned long clone_flags,
1168             unsigned long stack_start,
1169             struct pt_regs *regs,
1170             unsigned long stack_size,
1171             int __user *parent_tidptr,
1172             int __user *child_tidptr)
1173 {
1174     struct task_struct *p;
1175     int trace = 0;
1176     long pid;
1177
1178     if (unlikely(current->ptrace)) {
1179         trace = fork_traceflag (clone_flags);
1180         if (trace)
1181             clone_flags |= CLONE_PTRACE;
1182     }
1183
1184     p = copy_process(clone_flags, stack_start, regs, stack_size, parent_tidptr,
child_tidptr);
-----
```

第 1178~1183 行：这几行代码首先检查父进程是否允许新进程被跟踪。追踪功能在处理进程的函数中普遍使用。本书只大概介绍 `ptrace` 功能。为了确定子进程是否被跟踪，`fork_traceflag()` 必须检查 `clone_flags` 的值。如果在 `clone_flags` 中设置了 `CLONE_VFORK` 标志，并且 `SIGCHLD` 信号没有被父进程捕获，或者如果当前进程也设置了 `PT_TRACE_FORK` 标志，那么子进程就会被跟踪，当然，如果设置了 `CLONE_UNTRACED` 或 `CLONE_IDLETASK` 标志，就不会被跟踪了。

第 1184 行：这一行创建了新进程并且复制了寄存器的值。`copy_process()` 函数完成新进程空间的创建和描述符字段定义的大部分工作，但是新进程并不立即运行。要了解 `copy_process()` 的更多细节，围绕调度程序去解释才会更清楚。3.6 节将详细讨论相关细节。

```
-----
kernel/fork.c
...
1189     pid = IS_ERR(p) ? PTR_ERR(p) : p->pid;
1190
1191     if (!IS_ERR(p)) {
1192         struct completion vfork;
1193
1194         if (clone_flags & CLONE_VFORK) {
-----
```

```

1195         p->vfork_done = &vfork;
1196         init_completion(&vfork);
1197     }
1198
1199     if ((p->ptrace & PT_PTRACED) || (clone_flags & CLONE_STOPPED)) {
1200     ...
1203         sigaddset(&p->pending.signal, SIGSTOP);
1204         set_tsk_thread_flag(p, TIF_SIGPENDING);
1205     }
1206     ...

```

第 1189 行：这一行的代码是对指针错误的检查。如果发现指针出错，仅简单地返回指针错误。

第 1194~1197 行：这几行检查 do_fork() 是否被 vfork() 所调用。如果是，激活与 vfork() 相关的等待队列。

第 1199~1205 行：如果父进程被跟踪或者克隆操作被设置成 CLONE_STOPPED 标志，那么子进程在启动时将收到一个 SIGSTOP 信号，这样子进程就是以暂停状态启动的。

```

-----
kernel/fork.c
1207     if (!(clone_flags & CLONE_STOPPED)) {
1208     ...
1222         wake_up_forked_process(p);
1223     } else {
1224         int cpu = get_cpu();
1225
1226         p->state = TASK_STOPPED;
1227         if (!(clone_flags & CLONE_STOPPED))
1228             wake_up_forked_process(p); /* do this last */
1229         ++total_forks;
1230
1231         if (unlikely (trace)) {
1232             current->ptrace_message = pid;
1233             ptrace_notify ((trace << 8) | SIGTRAP);
1234         }
1235
1236         if (clone_flags & CLONE_VFORK) {
1237             wait_for_completion(&vfork);
1238             if (unlikely (current->ptrace & PT_TRACE_VFORK_DONE))
1239                 ptrace_notify ((PT_TRACE_EVENT_VFORK_DONE << 8) | SIGTRAP);
1240         } else
1241             ...
1248             set_need_resched();
1249     }
1250     return pid;
1251 }
-----

```

第 1226~1229 行：这几行将进程状态设置为 TASK_STOPPED。如果在 clone_flags 中未设置 CLONE_STOPPED 标志，那么就唤醒子进程；否则，让它等待一个唤醒信号。

第 1231~1234 行：如果在父进程上激活跟踪功能，则发送一个通知。

第 1236~1239 行：如果这是最初对 vfork() 的调用，那么就将父进程设置成阻塞状态，并发

送一个通知给跟踪者（如果父进程激活了跟踪功能的话）。这是通过把父进程放入等待队列中，并让它保持 `TASK_UNINTERRUPTIBLE` 状态直到子进程调用 `exit()` 或 `execve()` 来实现的。

第 1248 行：在当前进程（父进程）中设置 `need_resched`。这样就允许子进程先运行。

3.4 进程的生命周期

了解如何创建进程之后，有必要再了解一下在进程的整个生命周期中会发生什么。在此期间，进程总会发现自己处于各种状态。这些状态之间的转换既取决于进程所执行的操作，也取决于发送给进程的信号。本章给出的示例程序就处于 `TASK_INTERRUPTIBLE` 状态和 `TASK_RUNNING` 状态（其当前状态）。

进程所处的第一个状态是 `TASK_INTERRUPTIBLE`，这是在由 `do_fork()` 调用的 `copy_process()` 例程中创建进程时设置的。进程所处的第二个状态是 `TASK_RUNNING`，这是在退出 `do_fork()` 之前被设置的。这两个状态肯定会出现现在进程的生命周期中。接着会出现许多变化，这些变化决定进程处于何种状态。进程所处的最后一个状态是 `TASK_ZOMBIE`，这是在调用 `do_exit()` 期间被设置的。接下来看看进程的各种状态以及状态之间的转换方式，并解释进程是如何从一种状态转换到另一种状态的。

3.4.1 进程的状态

当进程正在运行时，意味着它的上下文已被装载到了 CPU 的寄存器和内存中，并且定义这个上下文的程序正在被执行。在某些特殊的时刻，有多种原因可能导致进程不能继续运行。这些原因有：进程正在等待还没有完成的输入；调度程序认为进程已经用完了分配给它的时间片，因此必须让其他进程运行。当进程能够运行却没有运行（比如重新调度）时被认为是就绪的，当等待输入时被认为是阻塞的。

图 3-10 显示了进程的抽象状态，每个抽象状态下列出了其对应的 Linux 可能的进程状态。表 3-5 给出了四种状态转换和引起转换的原因。表 3-6 将抽象状态和 Linux 内核中标识这些状态的值联系起来。

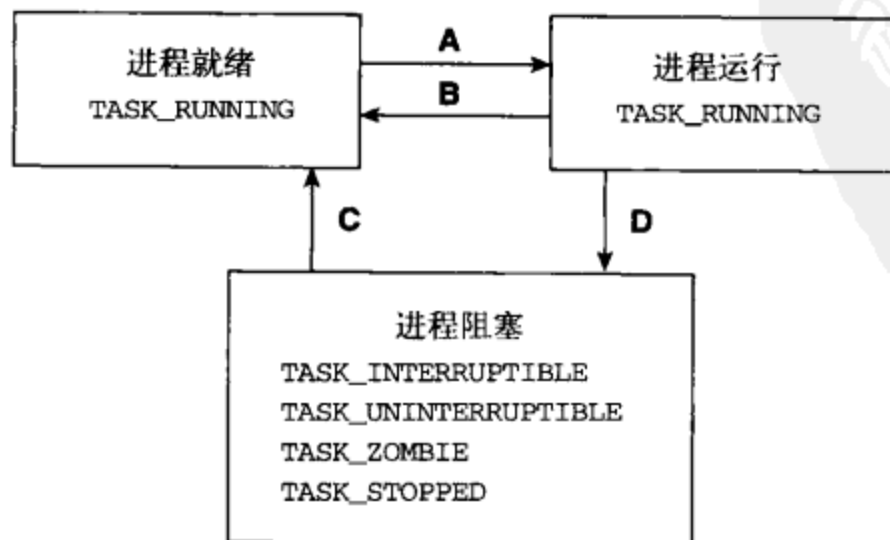


图 3-10 进程的状态转换

表3-5 进程的状态转换

转 换	原 因
就绪态→运行态 (A)	被调度程序选中
运行态→就绪态 (B)	时间片用完了 (被动的), 进程让出CPU (主动的)
阻塞态→就绪态 (C)	信号到来, 或等待的资源变为可用的
运行态→阻塞态 (D)	进程睡眠或等待事件

表3-6 Linux的标志和抽象进程状态间的关系

抽象状态	Linux的进程状态
就绪态	TASK_RUNNING
运行态	TASK_RUNNING
阻塞态	TASK_INTERRUPTIBLE TASK_UNINTERRUPTIBLE TASK_ZOMBIE TASK_STOPPED

注意 如果可以访问进程结构, `set_current_state()`就能直接设置其状态, 例如, 直接赋值设置 `current->state= TASK_INTERRUPTIBLE`。调用 `set_current_state(TASK_INTERRUPTIBLE)`可达到同样的效果。

3.4.2 进程状态的转换

现在来看看使进程从一种状态转换到另一种状态的各种事件。进程的抽象状态转换 (参阅表 3-5) 包括就绪态到运行态的转换、运行态到就绪态的转换、阻塞态到就绪态的转换和运行态到阻塞态的转换。每一种抽象状态转换至少对应一个 Linux 进程状态的转换。例如, 阻塞态到运行态的转换可以转化为 `TASK_INTERRUPTIBLE`、`TASK_UNINTERRUPTIBLE`、`TASK_ZOMBIE` 或 `TASK_STOPPED` 到 `TASK_RUNNING` 的转换。图 3-11 和表 3-7 详细描述了这些转换。

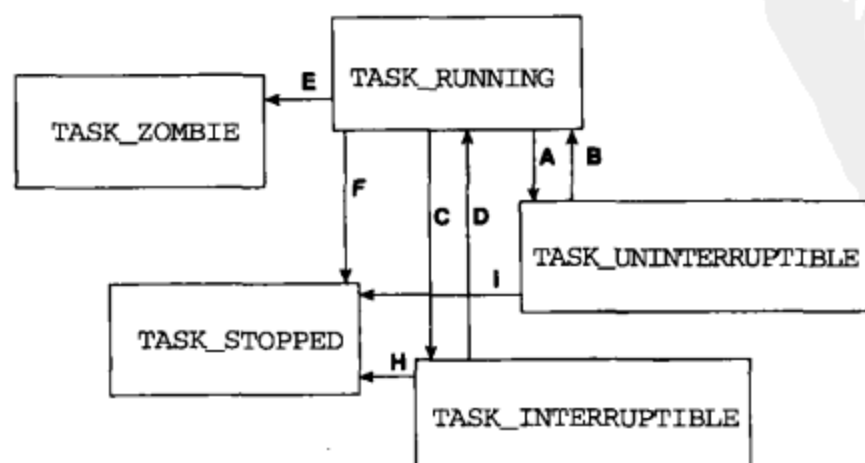


图 3-11 Linux 进程的状态转换

表3-7 Linux的进程状态转换

起始状态	结束状态	状态转换原因
TASK_RUNNING	TASK_UNINTERRUPTIBLE	进程进入等待队列
TASK_RUNNING	TASK_INTERRUPTIBLE	进程进入等待队列
TASK_RUNNING	TASK_STOPPED	进程接收到SIGSTOP信号., 或者进程被跟踪
TASK_RUNNING	TASK_ZOMBIE	进程被杀死, 但其父进程还没有调用sys_wait4()
TASK_INTERRUPTIBLE	TASK_STOPPED	收到信号时
TASK_UNINTERRUPTIBLE	TASK_STOPPED	被唤醒时
TASK_UNINTERRUPTIBLE	TASK_RUNNING	进程获得等待的资源时
TASK_INTERRUPTIBLE	TASK_RUNNING	进程获得等待的资源, 或收到信号后被设置为运行状态
TASK_RUNNING	TASK_RUNNING	由调度程序移入移出

接下来说明各种状态的转换, 并详细介绍普通进程转换类中的进程状态转换。

1. 从就绪态到运行态

“就绪态到运行态”的抽象状态转换并不对应实际的 Linux 进程状态转换, 因为进程的状态实际上没有发生变化 (进程仍然处于 TASK_RUNNING 状态)。然而, 进程确实进入了准备运行的队列 (或运行队列), 并真正由 CPU 执行。

● 从TASK_RUNNING到TASK_RUNNING

Linux 对当前正在使用 CPU 的进程并没有给出一个明确的状态, 即使进程已离开就绪队列, 并且它的上下文正在执行, 进程也一直保持着 TASK_RUNNING 状态。调度程序从运行队列中选择进程。在第 7 章将介绍调度程序如何选择下一个要运行的进程。

2. 从运行态到就绪态

在这种情况下, 即使进程本身发生了变化, 进程状态也不会改变。进程的抽象状态转换有助于读者理解究竟发生了什么。如前所述, 当进程从由 CPU 运行到进入运行队列时发生了状态转换, 经历了从运行到就绪的过程。

● 从TASK_RUNNING到TASK_RUNNING

由于 Linux 没有为其上下文由 CPU 执行的进程设置一个单独的状态, 因而此时进程不会经历一个明显的 Linux 状态转换, 还依然处于 TASK_RUNNING 状态。根据进程执行所花费时间的多少以及进程的优先级 (第 7 章将详细介绍), 调度程序决定何时将一个正在运行的进程切换下来放入运行队列。

3. 从运行态到阻塞态

当进程变为阻塞时, 它可能正处于如下某个状态: TASK_INTERRUPTIBLE、TASK_UNINTERRUPTIBLE、TASK_ZOMBIE 或 TASK_STOPPED, 如表 3-7 中所示, 接下来看看进程如何从 TASK_RUNNING 变成其中的每一个状态。

● TASK_RUNNING到TASK_INTERRUPTIBLE

这种状态通常是由 I/O 阻塞函数引起的, 因为这种函数必须等待一个事件或资源。对于一个

进程来说,处于 TASK_INTERRUPTIBLE 状态意味着什么呢?简单地说,进程肯定不在运行队列中,因为它还没有就绪。如果进程的资源变为可用(时间或硬件),或者如果一个信号到来,处于 TASK_INTERRUPTIBLE 状态的进程就会被唤醒。最初的系统调用的完成依赖于中断处理程序的实现。在示例程序中,子进程访问磁盘上的一个文件。磁盘驱动程序负责获知设备何时准备好供访问的数据。因此,驱动程序中有一部分类似于下面这段代码:

```
while(1)
{
    if(resource_available)
        break();
    set_current_state(TASK_INTERRUPTIBLE);
    schedule();
}
set_current_state(TASK_RUNNING);
```

当示例中的进程执行 open()调用时,它将进入 TASK_INTERRUPTIBLE 状态。此处,调用 schedule()把它从 CPU 上换下来,并选择运行队列的另外一个进程作为运行进程。在资源变为可用以后,该进程终止循环并把其状态设置为 TASK_RUNNING,这样该进程又被放回运行队列。进程在运行队列中等待,直到调度程序认为轮到它执行为止。

以下是函数 interruptible_sleep_on(),它能够将进程设置为 TASK_INTERRUPTIBLE 状态。

```
-----
kernel/sched.c
2504 void interruptible_sleep_on(wait_queue_head_t *q)
2505 {
2506     SLEEP_ON_VAR
2507
2508     current->state = TASK_INTERRUPTIBLE;
2509
2510     SLEEP_ON_HEAD
2511     schedule();
2512     SLEEP_ON_TAIL
2513 }
-----
```

SLEEP_ON_HEAD 宏和 SLEEP_ON_TAIL 宏分别完成向等待队列添加和删除一个进程(参见 3.7 节)的操作。SLEEP_ON_VAR 宏初始化进程的等待队列入口,使得可以向等待队列添加进程。

● 从 TASK_RUNNING 到 TASK_UNINTERRUPTIBLE

除了当进程处于内核态时不理睬到来的信号外, TASK_UNINTERRUPTIBLE 状态与 TASK_INTERRUPTIBLE 状态类似。这个状态也是在调用 do_fork()创建进程时,为进程设置的默认状态。函数 sleep_on()可以将进程设置为 TASK_UNINTERRUPTIBLE 状态。

```
-----
kernel/sched.c
2545 long fastcall __sched sleep_on(wait_queue_head_t *q)
2546 {
2547     SLEEP_ON_VAR
2548
2549     current->state = TASK_UNINTERRUPTIBLE;
```



```

2550
2551     SLEEP_ON_HEAD
2552     schedule();
2553     SLEEP_ON_TAIL
2554
2555     return timeout;
2556 }

```

该函数把进程放入等待队列，设置进程状态，并且调用调度程序。

- 从TASK_RUNNING到TASK_ZOMBIE

处于 TASK_ZOMBIE 状态的进程叫作**僵死进程**。每个进程在它的生命周期中都要经历这个状态。进程处于这种状态的时长依赖于它的父进程。为了更好地理解这一点，联想一下，在 UNIX 系统中，任何进程都可以通过调用 wait() 或 waitpid()（详细内容请参见 3.5.3 节）来获取子进程的退出状态。因此，即使子进程已经终止了，也还需要为父进程保留少量信息。仅仅因为父进程需要了解子进程的状态就让它一直活着，显然是很奢侈的事情；因此，进程有一种状态，除了进程描述符仍被保留外，其他资源都被释放并回收，这就是僵死状态。

设置 TASK_ZOMBIE 这个临时状态，是在调用 sys_exit()（详细内容参见第 3.5 节）时进行的。处于这种状态的进程不会再运行，只能转换到 TASK_STOPPED 状态。

如果一个进程处于僵死状态的时间过长，它的父进程将不会回收它。系统不能杀死僵死进程，因为它实际上就是死的。这意味着不存在杀死的僵死进程，只有等待被释放的进程描述符。

- 从TASK_RUNNING到TASK_STOPPED

这个转换会在两种情况下出现。第一种情况是正在被调试器或跟踪程序处理的进程；第二种情况是进程收到 SIGSTOP 或某种停止信号。

- 从TASK_UNINTERRUPTIBLE或者TASK_INTERRUPTIBLE到TASK_STOPPED

TASK_STOPPED 管理 SMP 系统中的进程或正在处理信号的进程。当进程收到一个唤醒信号时，或者内核明确需要进程不响应任何事情时（例如，如果进程被置为 TASK_INTERRUPTIBLE 状态，它是会响应其他事情的），进程就被设置为 TASK_STOPPED 状态。

与处于 TASK_ZOMBIE 状态的进程不同，处于 TASK_STOPPED 状态的进程仍然能接收 SIGKILL 信号。

4. 从阻塞态到就绪态

当进程得到等待的数据或硬件时，将从阻塞态转换到就绪态。这一状态转换对应着 Linux 的两种状态转换，分别是 TASK_INTERRUPTIBLE 到 TASK_RUNNING 和从 TASK_UNINTERRUPTIBLE 到 TASK_RUNNING。

3.5 进程的终止

进程可以显式、主动地终止，也可以隐式、主动地终止，或者被动终止。主动终止可通过两种途径来实现。

- (1) 从 main() 函数返回（隐式的）。
- (2) 调用 exit()（显式的）。

从主函数执行一个返回实际上转化为对 `exit()` 的调用。链接程序在这类情况下会引入对 `exit()` 的调用。

进程的被动终止可以通过三种途径来实现。

- (1) 进程可能收到自己不能处理的一个信号。
- (2) 当进程在内核态执行时可能产生一个异常。
- (3) 进程可能收到了 `SIGABRT` 信号或其他终止信号。

对进程的终止如何处理依赖于父进程是否消亡。一个进程可能：

- 先于父进程终止；
- 在父进程之后终止。

在第一种情况下，子进程被变成一个僵死进程，直到父进程调用 `wait()` 或 `waitpid()`。在第二种情况下，`init()` 进程将成为子进程的新父进程。当任何进程终止的时候，内核都要查看一遍所有活着的进程，核实将要终止的进程是否是某个活着进程的父进程，如果是，内核就把其子进程的父进程 PID 修改为 1。

接下来再来看一下示例程序，并且跟踪它直到消亡。该进程显式地调用了 `exit(0)`。（注意，调用 `_exit()`、`return(0)` 或者从 `main` 函数的结束处退出也是一样的。）C 库函数 `exit()` 接着调用 `sys_exit()` 系统调用。接下来分析下面的代码，看看从这里开始对进程做了些什么。

首先，来看看终止进程的函数。如前所述，进程 `foo` 调用了 `exit()`，`exit()` 接着调用了第一个函数——`sys_exit()`。我们将追踪 `sys_exit()` 的调用，并且一直深入到 `do_exit()` 的内部。

3.5.1 `sys_exit()` 函数

```
-----
kernel/exit.c
asmlinkage long sys_exit(int error_code)
{
    do_exit((error_code&0xff)<<8);
}
-----
```

`sys_exit()` 不随体系结构而变化，它的工作也相当简单，仅仅是把退出码转换成内核要求的格式并调用 `do_exit()`。

3.5.2 `do_exit()` 函数

```
-----
kernel/exit.c
707 NORET_TYPE void do_exit(long code)
708 {
709     struct task_struct *tsk = current;
710
711     if (unlikely(in_interrupt()))
712         panic("Aieee, killing interrupt handler!");
713     if (unlikely(!tsk->pid))
714         panic("Attempted to kill the idle task!");
715     if (unlikely(tsk->pid == 1))
-----
```

```

716     panic("Attempted to kill init!");
717     if (tsk->io_context)
718         exit_io_context();
719     tsk->flags |= PF_EXITING;
720     del_timer_sync(&tsk->real_timer);
721
722     if (unlikely(in_atomic()))
723         printk(KERN_INFO "note: %s[%d] exited with preempt_count %d\n",
724             current->comm, current->pid,
725             preempt_count());
-----

```

第 707 行：参数 code 包含了进程返回给父进程的退出代码。

第 711~716 行：排除一些无效但可能发生的情况，包括以下两种情况。

(1) 确定退出进程没有处于中断处理中。

(2) 确保退出进程不是 idle 进程 (PID=0) 或 init 进程 (PID=1)。注意，init 进程只有在系统关闭时才能被杀死。

第 719 行：在此，将进程的进程结构的 flag 字段设置为 PF_EXITING。这表明进程正在关闭。例如，为一个给定进程创建间隔定时器时要使用该标志。检查进程的标志是否设置有助于防止多余的处理。

```

-----
kernel/exit.c
...
727     profile_exit_task(tsk);
728
729     if (unlikely(current->ptrace & PT_TRACE_EXIT)) {
730         current->ptrace_message = code;
731         ptrace_notify((PTRACE_EVENT_EXIT << 8) | SIGTRAP);
732     }
733
734     acct_process(code);
735     __exit_mm(tsk);
736
737     exit_sem(tsk);
738     __exit_files(tsk);
739     __exit_fs(tsk);
740     exit_namespace(tsk);
741     exit_thread();
...
-----

```

第 729~732 行：如果进程正在被追踪并且已经设置了 PT_TRACE_EXIT 标志，那么就传递退出代码并通知父进程。

第 735~742 行：这几行清理和回收进程已经使用过并且以后不再使用的资源。__exit_mm() 释放分配给进程的内存，并释放与进程相关的 mm_struct。exit_sem() 把进程从所有 IPC 信号量中分离出来。__exit_files() 释放分配给进程的所有文件，减小文件描述符的计数。__exit_fs() 释放所有的文件系统数据。

```

-----
kernel/exit.c
...
744  if (tsk->leader)
745      disassociate_ctty(1);
746
747  module_put(tsk->thread_info->exec_domain->module);
748  if (tsk->binfmt)
749      module_put(tsk->binfmt->module);
...
-----

```

第 744~745 行：如果进程是一个会话首进程，它极可能拥有一个控制终端或 `tty`。这个函数把会话首进程从它的控制终端 `tty` 中分离出来。

第 747~749 行：这几行减少模块的引用数。

```

-----
kernel/exit.c
...
751  tsk->exit_code = code;
752  exit_notify(tsk);
753
754  if (tsk->exit_signal == -1 && tsk->ptrace == 0)
755      release_task(tsk);
756
757  schedule();
758  BUG();
759  /* Avoid "noreturn function does return". */
760  for (;;) ;
761  }
...
-----

```

第 751 行：在 `task_struct` 的 `exit_code` 字段中设置退出代码。

第 752 行：给父进程发送 `SIGCHLD` 信号，并设置进程状态为 `TASK_ZOMBIE`。`exit_notify()` 向垂死进程的亲属通报进程的消亡。此时，该进程的父进程将接到退出代码的通知，而其子进程则将自己的父进程设置为 `init` 进程。唯一例外的是，如果该进程所在的组中还有其他进程存在，那么现存的这个进程就作为其子进程的代理父进程。

第 754 行：如果 `exit_signal` 是 `-1`（表明有一个错误），并且进程没有被跟踪，那么内核将调用调度程序去释放这个进程的描述符并回收它的时间片。

第 757 行：将处理器让给新进程。在第 7 章中将看到，调用 `schedule()` 不会有返回值。此后的所有代码捕获不太可能发生的情况，或者消除编译器的警告。

3.5.3 通知父进程和 `sys_wait4()`

当一个进程终止时，要通知它的父进程。在这之前，进程处于僵死状态，在此状态中，除了进程描述符仍然保留着以外，它的所有资源都已归还内核。当子进程终止时，父进程（例如 `Bash shell`）收到内核发送给它的 `SIGCHLD` 信号。本例中，当 `shell` 想要收到通知时就调用 `wait()`。父

进程可以通过不执行中断处理程序来忽略信号，也可以改为选择在任何时刻调用 `wait()` (或 `waitpid()`)。

`wait` 函数族扮演以下两种常见的角色。

❑ 殡仪业者。获知进程消亡的消息。

❑ 掘墓人。消除进程的所有痕迹。

父进程可以选择调用 `wait` 函数族中的 4 个函数之一：

❑ `pid_t wait(int *status)`

❑ `pid_t waitpid(pid_t pid, int *status, int options)`

❑ `pid_t wait3(int *status, int options, struct rusage *rusage)`

❑ `pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage)`

每个函数又依次调用 `sys_wait4()`，大部分的通知会在 `sys_wait4()` 中产生。

如果进程调用 `wait` 函数族中的某一函数，那么它会被阻塞，直到它的某个子进程终止，或者如果子进程已经终止（或其父进程已经没有其他子进程），该进程就立即返回。`sys_wait4()` 函数显示内核如何管理这个通知：

```
-----
kernel/exit.c
1031  asmlinkage long sys_wait4(pid_t pid,unsigned int * stat_addr,
int options, struct rusage * ru)
1032  {
1033    DECLARE_WAITQUEUE(wait, current);
1034    struct task_struct *tsk;
1035    int flag, retval;
1036
1037    if (options & ~(WNOHANG|WUNTRACED|__WNOTHREAD|__WCLONE|__WALL))
1038      return -EINVAL;
1039
1040    add_wait_queue(&current->wait_chldexit,&wait);
1041    repeat:
1042      flag = 0;
1043      current->state = TASK_INTERRUPTIBLE;
1044      read_lock(&tasklist_lock);
1045      ...
-----
```

第 1031 行：这些参数包括目标进程的 PID、存放子进程退出状态的地址、传给 `sys_wait4()` 的标志以及存放子进程资源使用信息的地址。

第 1033 和 1040 行：声明一个等待队列，并把进程加入该队列。（详细内容参见 3.7 节。）

第 1037~1038 行：这两行代码主要检查错误条件。如果传给 `sys_wait4()` 的参数 `options` 是无效的，函数将返回一个错误码。在本例中，返回错误码 `EINVAL`。

第 1042 行：`flag` 变量的初始值被设置为 0。一旦发现参数 `pid` 和调用进程的某个子进程的 `pid` 匹配，就要修改这个变量。

第 1043 行：将调用进程设置为阻塞状态，进程的状态从 `TASK_RUNNING` 变为 `TASK_INTERRUPTIBLE`。

```

-----
kernel/exit.c
...
1045     tsk = current;
1046     do {
1047         struct task_struct *p;
1048         struct list_head *_p;
1049         int ret;
1050
1051         list_for_each(_p, &tsk->children) {
1052             p = list_entry(_p, struct task_struct, sibling);
1053
1054             ret = eligible_child(pid, options, p);
1055             if (!ret)
1056                 continue;
1057             flag = 1;
1058             switch (p->state) {
1059                 case TASK_STOPPED:
1060                     if (!(options & WUNTRACED) &&
1061                         !(p->ptrace & PT_PTRACED))
1062                         continue;
1063                     retval = wait_task_stopped(p, ret == 2,
1064                         stat_addr, ru);
1065                     if (retval != 0) /* He released the lock. */
1066                         goto end_wait4;
1067                     break;
1068                 case TASK_ZOMBIE:
1069
1070                     if (ret == 2)
1071                         continue;
1072                     retval = wait_task_zombie(p, stat_addr, ru);
1073                     if (retval != 0) /* He released the lock. */
1074                         goto end_wait4;
1075                     break;
1076             }
1077         }
1078     }
1079
1080     ...
1091     tsk = next_thread(tsk);
1092     if (tsk->signal != current->signal)
1093         BUG();
1094 } while (tsk != current);
...
-----

```

第 1046 行和 1094 行：当 do while 循环检测到自己还是当前进程时只执行一次循环，当检测到是其他进程时便继续循环。

第 1051 行：对进程的子进程列表中的每个进程重复操作。请注意，是父进程在等待子进程的退出。父进程当前处于 TASK_INTERRUPTIBLE 状态，正在逐个扫描其子进程。

第 1054 行：确定被传递的 pid 参数是否不合理。

第 1058~1079 行：检查进程的每个子进程的状态。仅当其子进程停止或僵死时才进行这种处理，如果进程正在睡眠、就绪或运行（其余状态），则什么也不做。如果子进程是 TASK_STOPPED 状态，并且使用了 UNTRACED 选项（这意味着进程由于跟踪而未被停止），则核实是否已经报告了子进程的状态，并且返回子进程的信息。如果子进程是 TASK_ZOMBIE 状态，则撤销它。


```

-----
kernel/exit.c
...
1106   retval = -ECHILD;
1107   end_wait4:
1108   current->state = TASK_RUNNING;
1109   remove_wait_queue(&current->wait_chldexit,&wait);
1110   return retval;
1111 }
-----

```

第 1106 行：如果执行到了这里，说明参数所指定的 PID 不是所调用进程的子进程。ECHILD 错误用于通报这种事件。

第 1107~1111 行：至此，子进程列表处理完毕，所有需要撤销的子进程都已被撤销。父进程的阻塞被解除，它的状态又被设置为 TASK_RUNNING。最后，等待队列也被删除。

至此，读者应该对进程在它的生命周期中所经历的各种状态、完成状态转换的内核函数，以及内核用于记录所有这些信息的数据结构都很熟悉了。接下来再看一看调度程序如何控制和管理进程以达到多线程系统的效果。当然也会看到进程状态转换的更多细节。

3.6 了解进程的动态：调度程序的基本构架

迄今为止，我们了解了以进程为中心的状态和状态转换，但还没有谈及如何管理进程状态的转换，也没有讨论执行进程的运行和停止操作的内核基本框架结构。调度程序将处理所有这些细节。完成了对进程生命周期的探索后，现在来介绍调度程序的基本知识，以及进程创建期间调度程序如何与函数 `do_fork()` 交互。

3.6.1 基本结构

调度程序操作的对象是一个称为运行队列的结构。系统中的每个 CPU 都有一个运行队列。运行队列中的核心数据结构是两个按优先级排序的数组。其中一个包含了活跃进程，另外一个包含了到期进程。通常，一个活跃进程运行一段固定的时间（时间片长度或时间片），然后被插入到期数组去等待更多的 CPU 时间。当活跃数组为空时，调度程序通过交换活跃数组和到期数组的指针来交换两个数组。然后，调度程序开始执行新活跃数组中的进程。

图 3-12 说明了运行队列中的优先级数组。优先级数组结构的定义如下：

```

-----
kernel/sched.c
192   struct prio_array {
193     int nr_active;
194     unsigned long bitmap[BITMAP_SIZE];
195     struct list_head queue[MAX_PRIO];
196   };
-----

```

`prio_array` 结构的字段如下所示。

□ **nr_active**。计数器，记录优先级数组中的进程数。

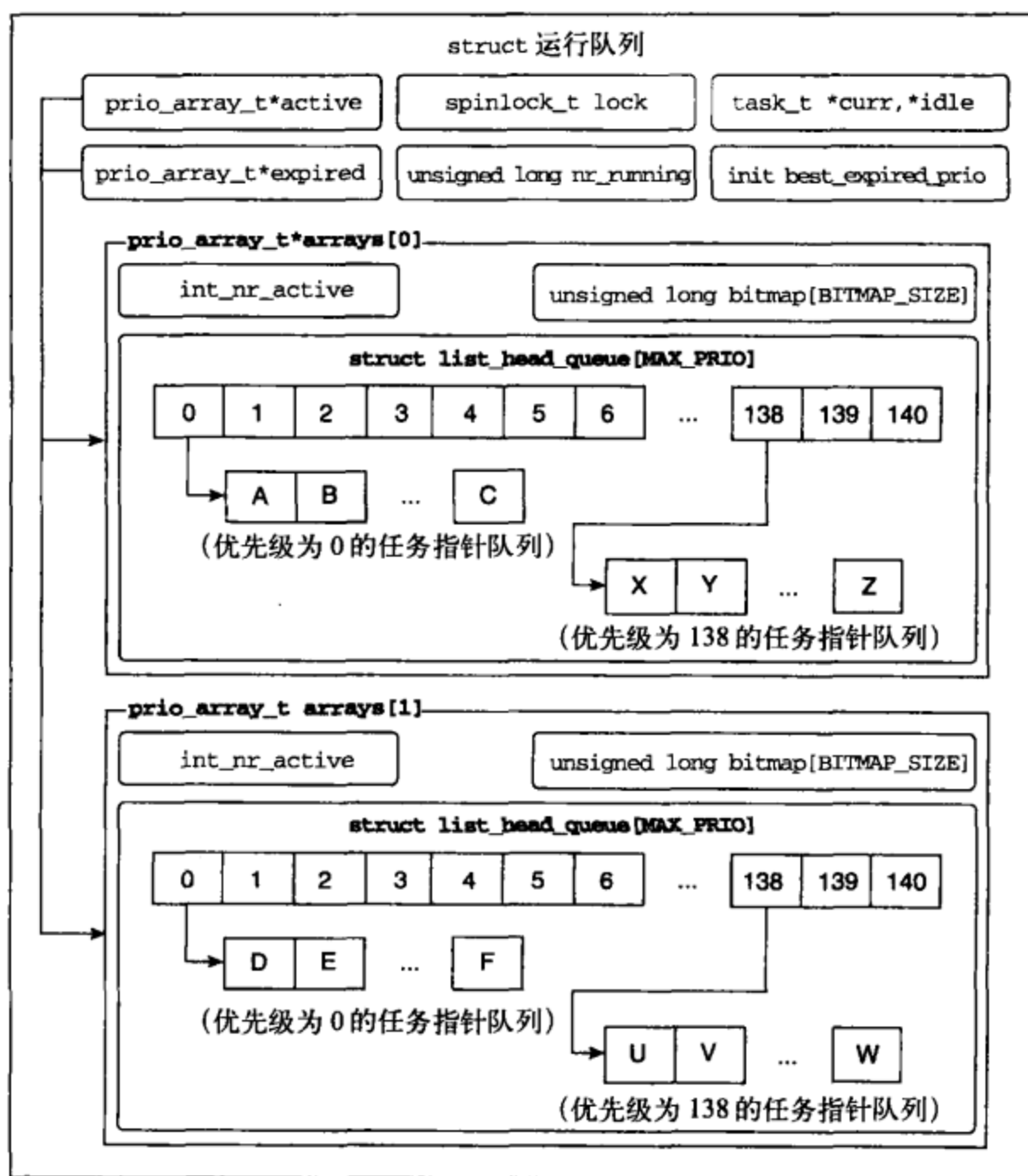


图 3-12 运行队列中的优先级数组

- **bitmap**。记录数组中的优先级。bitmap 的实际长度取决于系统中无符号长整型的大小。它始终足够存放 MAX_PRIO 个位，但也可能会更长。
- **queue**。存储进程链表的数组。每个链表存放特定优先级的进程，因此，queue[0]就是存放所有优先级为 0 的进程的链表，queue[1]就是存放所有优先级为 1 的进程的链表，等等。

在基本理解了运行队列是如何组织的之后，再通过单 CPU 系统上的调度程序来着手理解进程。

3.6.2 从等待中醒来或者激活

回想一下，当进程调用 `fork()` 时，就创建了新的进程。如前所述，调用 `fork()` 的进程叫做父进程，新的进程叫做子进程。新创建的进程需要被调度才能访问 CPU，这是通过 `do_fork()` 函数来完成的。

`do_fork()`中有两行用于处理调度程序的重要代码，这两行代码同时还与唤醒进程有关。在 `linux/kernel/fork.c` 的第 1 184 行调用了函数 `copy_process()`，该函数将调用函数 `sched_fork()`，以便把进程初始化为即将插入调度程序运行队列的状态。在 `linux/kernel/fork.c` 的第 1 222 行调用了函数 `wake_up_forked_process()`，它用于获取已初始化的进程，并把它真正插入运行队列。考虑到新进程可能被杀死，初始化操作和插入操作被分开进行，否则的话，进程可能还没有被调度就已经被终止了。只有创建成功、初始化成功且没有挂起的信号的进程才能被调度。

`sched_fork()`：调度程序初始化新创建的进程

调度程序需要初始化新创建的进程，`sched_fork()`函数就完成这种基础结构的设置。

```
-----
kernel/sched.c
719 void sched_fork(task_t *p)
720 {
721     /*
722      * We mark the process as running here, but have not actually
723      * inserted it onto the runqueue yet. This guarantees that
724      * nobody will actually run it, and a signal or other external
725      * event cannot wake it up and insert it on the runqueue either.
726      */
727     p->state = TASK_RUNNING;
728     INIT_LIST_HEAD(&p->run_list);
729     p->array = NULL;
730     spin_lock_init(&p->switch_lock);
-----
```

第 727 行：在 `do_fork()` 和 `copy_process()` 对进程是否正确创建进行验证之前，就把进程标记为运行状态（这是通过把进程结构的 `State` 属性设置为 `TASK_RUNNING` 来实现的），以确保没有其他事件能把该进程插入运行队列并运行它。当这些验证都通过以后，`do_fork()`才调用 `wake_up_forked_process()`把进程真正添加到运行队列中。

第 728~730 行：初始化进程的 `run_list` 字段。当进程被激活时，它的 `run_list` 字段被链接到运行队列中某个优先级数组的队列结构中。把进程的 `array` 域置为 `NULL`，表示进程不在运行队列的任何一个优先级数组中。`sched_fork()`的下一部分代码（第 731~739 行）处理内核的抢占（关于抢占的详细内容参见第 7 章）。

```
-----
kernel/sched.c
740     /*
741      * Share the timeslice between parent and child, thus the
742      * total amount of pending timeslices in the system doesn't change,
743      * resulting in more scheduling fairness.
744      */
745     local_irq_disable();
746     p->time_slice = (current->time_slice + 1) >> 1;
747     /*
748      * The remainder of the first timeslice might be recovered by
749      * the parent if the child exits early enough.
750      */
751     p->first_time_slice = 1;
-----
```

```

752  current->time_slice >>= 1;
753  p->timestamp = sched_clock();
754  if (!current->time_slice) {
755      /*
756       * This case is rare, it happens when the parent has only
757       * a single jiffy left from its timeslice. Taking the
758       * runqueue lock is not a problem.
759       */
760      current->time_slice = 1;
761      preempt_disable();
762      scheduler_tick(0, 0);
763      local_irq_enable();
764      preempt_enable();
765  } else
766      local_irq_enable();
767 }

```

第 740~753 行：在禁用本地中断之后，用移位运算符给子进程划分父进程的一部分时间片。新进程的第一个时间片被设置为 1，因为它还未被运行，其时间戳被初始化成以纳秒为单位的当前时间。

第 754~767 行：如果父进程的时间片是 1，这种划分将导致父进程没有剩余的时间运行。因为父进程是调度程序的当前进程，因此需要调度程序选择一个新的进程来运行。这可以通过调用 `scheduler_tick()`（见第 762 行）来完成。为了确保调度程序不受干扰地选择一个新进程，应该禁止内核被抢占。调度操作完成以后，则启用抢占并恢复本地中断。

此处，新创建的进程初始化其与调度程序相关的变量，并让其初始时间片为父进程剩余时间片的一半。内核强迫进程牺牲分配给它的一部分 CPU 时间，并把这些时间分给其子进程来防止进程占有大块的处理器时间。如果进程被给予一个固定不变的时间片，恶意进程可能会生出许多子进程，从而快速而贪婪地占有 CPU。

进程被成功地初始化并且初始化被验证有效以后，`do_fork()` 就调用 `wake_up_forked_process()`：

```

-----
kernel/sched.c
922 /*
923  * wake_up_forked_process - wake up a freshly forked process.
924  *
925  * This function will do some initial scheduler statistics housekeeping
926  * that must be done for every newly created process.
927  */
928 void fastcall wake_up_forked_process(task_t * p)
929 {
930     unsigned long flags;
931     runqueue_t *rq = task_rq_lock(current, &flags);
932
933     BUG_ON(p->state != TASK_RUNNING);
934
935     /*
936     * We decrease the sleep average of forking parents

```

```

937  * and children as well, to keep max-interactive tasks
938  * from forking tasks that are max-interactive.
939  */
940  current->sleep_avg = JIFFIES_TO_NS(CURRENT_BONUS(current) *
941    PARENT_PENALTY / 100 * MAX_SLEEP_AVG / MAX_BONUS);
942
943  p->sleep_avg = JIFFIES_TO_NS(CURRENT_BONUS(p) *
944    CHILD_PENALTY / 100 * MAX_SLEEP_AVG / MAX_BONUS);
945
946  p->interactive_credit = 0;
947
948  p->prio = effective_prio(p);
949  set_task_cpu(p, smp_processor_id());
950
951  if (unlikely(!current->array))
952    __activate_task(p, rq);
953  else {
954    p->prio = current->prio;
955    list_add_tail(&p->run_list, &current->run_list);
956    p->array = current->array;
957    p->array->nr_active++;
958    rq->nr_running++;
959  }
960  task_rq_unlock(rq, &flags);
961  }

```

第930~934行：调度程序做的第一件事就是锁住运行队列结构。对运行队列的任何更改必须在上锁情况下进行。如果进程没有被标记为 TASK_RUNNING（通过 sched_fork() 初始化后应该为 TASK_RUNNING，参见前面列出的 kernel/sched.c 中的第727行），那么就抛出一个 bug 通知。

第940~947行：调度程序计算父进程和子进程的睡眠平均值。睡眠平均值是进程睡眠所花时间和进程运行所花时间的比。它随着进程睡眠时间数而增加，随着进程运行时每个定时器的节拍而减少。交互式进程或者 I/O 密集型（I/O bound）进程在等待输入上花费了大部分时间，它们的睡眠平均值通常很高。非交互式进程或者 CPU 密集型（CPU-bound）进程在使用 CPU（而不是等待 I/O）上花费了大部分时间，这些进程的睡眠平均值很低。因为用户想看到他们的输入结果，例如敲击键盘或移动鼠标，因此交互式进程被赋予了比非交互式进程更多的调度优势。具体而言，在交互式进程的时间片到期以后，调度程序把它重新插入到活跃的优先级数组中。为防止交互式进程创建一个非交互的子进程，从而占有一个不相称的 CPU 份额，这些公式用来降低父进程和子进程的睡眠平均值。如果新创建的进程是交互式的，它立刻去睡眠足够长的时间，重新获得本已失去的所有调度优势。

第948行：函数 effective_prio() 修改进程的静态优先级。它返回一个 100~139（MAX_RT_PRIO 到 MAX__PRIO-1）的优先级。基于进程以前使用 CPU 的情况以及用于睡眠的时间，进程的静态优先级可以增加或减少一个不大于 5 的数，但总保持在刚才提到的范围内。我们已经从命令行开始讨论了进程的 nice 值，它可以在 +19~20 的范围内变化（最低优先级至最高优先级）。nice 优先级 0 对应静态优先级的 120。

第 949 行：把进程的 CPU 属性设置成当前 CPU。

第 951~960 行：总的来看，这段代码是新进程（或者说子进程）从当前进程——它的父进程复制调度信息，然后把自己插入运行队列的适当位置。至此，完成了对运行队列的修改，因此将它解锁。下面的段落和图 3-13 将更详细地讨论这个进程。

指针 `array` 指向运行队列中的优先级数组。如果当前进程没有指向一个优先级数组，那就意味着当前进程已经完成或者正在睡眠。如果是那样的话，当前进程的 `runlist` 字段就不在运行队列的优先级数组的队列中，这意味着 `list_add_tail()`（第 955 行）操作会失败。因此改为利用 `__activate_task()` 来插入新创建的进程，这个函数就把新进程添加到运行队列中而无需涉及它的父进程。

通常，当前进程正在运行队列中等待 CPU 时间时，进程被添加到优先级数组中处于 `p->prio` 位置的队列。添加了新进程的数组其进程计数器 `nr_active` 增加 1，运行队列也让它的进程计数器 `nr_running` 增加 1。最后，将运行队列解锁。

当前进程不指向运行队列中的优先级数组时，这对于观察调度程序如何管理运行队列和优先级数组的属性是很有用处的。

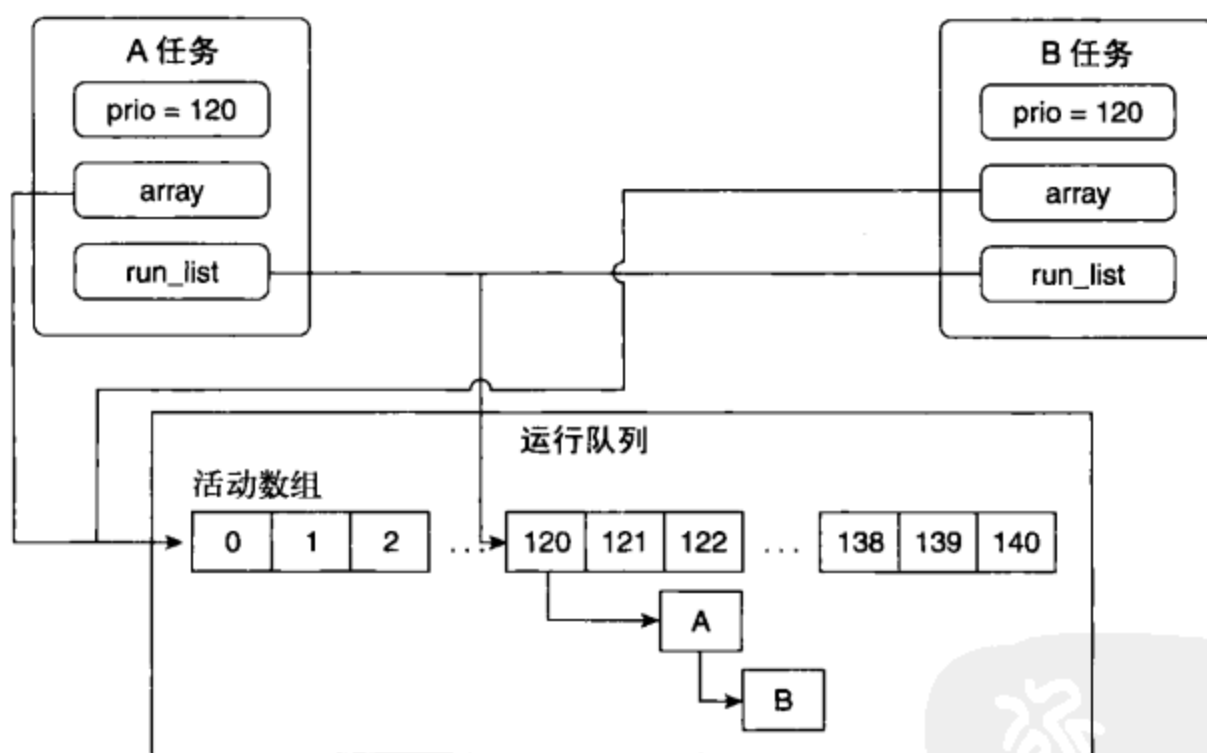


图 3-13 运行队列的插入

```
-----
kernel/sched.c
366 static inline void __activate_task(task_t *p, runqueue_t *rq)
367 {
368     enqueue_task(p, rq->active);
369     rq->nr_running++;
370 }
-----
```

`__activate_task()` 把给定进程 `p` 放到运行队列 `rq` 中的活跃优先级数组上，并且把运行队列的 `nr_running` 字段增加 1，`nr_running` 是一个表示运行队列上进程总数的计数器。


```

-----
kernel/sched.c
311 static void enqueue_task(struct task_struct *p, prio_array_t *array)
312 {
313     list_add_tail(&p->run_list, array->queue + p->prio);
314     __set_bit(p->prio, array->bitmap);
315     array->nr_active++;
316     p->array = array;
317 }
-----

```

第 311~312 行：当初始化优先级数组时，`enqueue_task()` 获得进程 `p`，并把它放到优先级数组 `array` 上。

第 313 行：进程的 `run_list` 被添加到优先级数组中位于 `p->prio` 处队列的队尾。

第 314 行：设置优先级为 `p->prio` 的优先级数组位图，当调度程序运行时，就能够看到一个进程以优先级 `p->prio` 运行。

第 315 行：增加优先级数组的进程计数器，以反映加入了新进程。

第 316 行：设置进程的数组指针 `p->array`，使之指向刚添加了进程的优先级数组。

简而言之，即使由于整个调度程序中很多类似的名字使得代码容易让人混淆，添加一个新创建进程的操作还是相当简单的。进程被放置在运行队列优先级数组中某个链表的末尾，该链表在数组中的位置是由进程的优先级来确定的。然后，进程在它的数据结构中记录优先级数组的位置和它所在链表的位置。

3.7 等待队列

前面已经讨论了 `TASK_RUNNING` 状态和 `TASK_INTERRUPTIBLE` 或 `TASK_UNINTERRUPTIBLE` 状态之间的转换。现在来看看涉及这一状态转换的另一个结构。当进程等待一个外部事件发生时，就把它从运行队列删除并放到等待队列上。等待队列是一个 `wait_queue_t` 结构的双向链表。设置 `wait_queue_t` 结构是为了保存记录等待进程所需的所有信息。等待一个特定外部事件的所有进程被放到同一个等待队列中。当某个等待队列上的进程被唤醒时，该进程核实它所等待的条件，然后或者继续睡眠，或者从等待队列删除自己并把自己设置回 `TASK_RUNNING`。回想一下，当父进程想要获知它所创建的子进程的状态时，`sys_wait4()` 系统调用使用了等待队列。注意，等待外部事件的进程（因此不再处于运行队列^①中）可能处于 `TASK_INTERRUPTIBLE` 状态，也可能处于 `TASK_UNINTERRUPTIBLE` 状态。

等待队列是 `wait_queue_t` 结构的双向链表，`wait_queue_t` 结构中有指向阻塞进程 `task` 结构的指针。每个链表以 `wait_queue_head_t` 结构打头，该结构标记链表的头部，并存放 `wait_queue_t` 链表的自旋锁，自旋锁可以防止 `wait_queue_t` 的额外竞争条件。图 3-14 说明了等待队列是如何实现的。现在来看看 `wait_queue_t` 和 `wait_queue_head_t` 结构。

```

-----
include/linux/wait.h

```

① 进程睡眠以后，它就被从运行队列删除，从而为另外一个进程让出 CPU 的控制权。

```

19 typedef struct __wait_queue wait_queue_t;
...
23 struct __wait_queue {
24     unsigned int flags;
25     #define WQ_FLAG_EXCLUSIVE 0x01
26     struct task_struct * task;
27     wait_queue_func_t func;
28     struct list_head task_list;
29 };
30
31 struct __wait_queue_head {
32     spinlock_t lock;
33     struct list_head task_list;
34 };
35 typedef struct __wait_queue_head wait_queue_head_t;

```

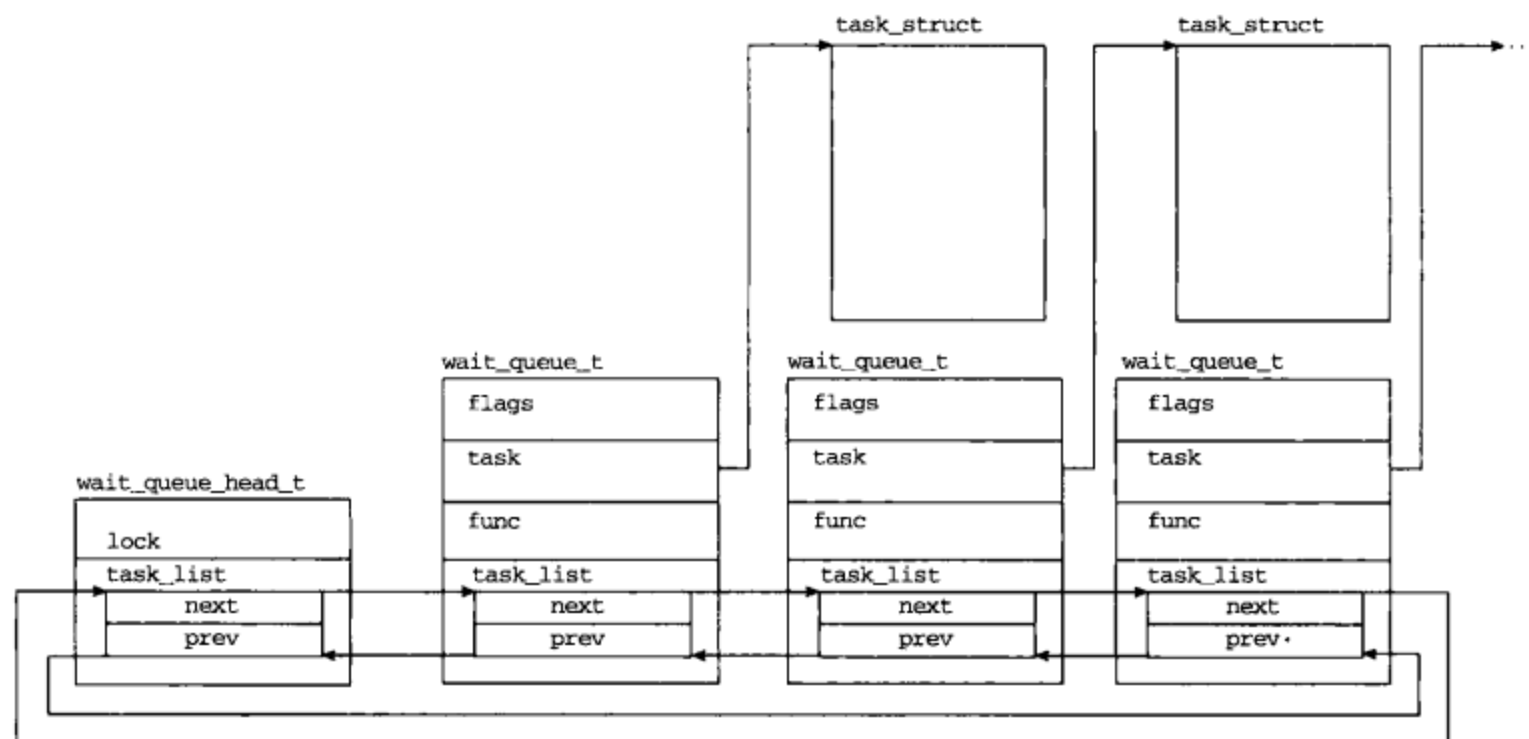


图 3-14 等待队列的结构

wait_queue_t 结构由下列字段组成。

- ❑ **flags**。可以存放值 WQ_FLAG_EXCLUSIVE (被设置为 1) 或 ~WQ_FLAG_EXCLUSIVE (被设置为 0)。WQ_FLAG_EXCLUSIVE 标志标记这个进程为独占式进程。独占式进程和非独占式进程将在下一节中讨论。
- ❑ **task**。一个指针，指向被加入到等待队列上的进程的进程描述符。
- ❑ **func**。存放函数的一个结构，该函数用于唤醒等待队列上的进程。这个字段的默认值为 default_wake_function()，该函数将在 3.7.2 节中详细介绍。

wait_queue_func_t 定义如下：

```

-----
include/linux/wait.h
typedef int (*wait_queue_func_t)(wait_queue_t *wait,
unsigned mode, int sync);
-----

```

此处, `wait` 是指向等待队列的指针, 而 `mode` 是 `TASK_INTERRUPTIBLE` 或者 `TASK_UNINTERRUPTIBLE`, `sync` 表示唤醒是否应被同步。

□ **task_list**。这个结构包含了两个指针, 分别指向等待队列中的前一个进程和后一个进程。

`__wait_queue_head` 结构是等待队列链表的表头, 由下列字段组成:

□ **lock**。每个链表有一个锁, 这使得向等待队列添加或删除数据项时能够同步。

□ **task_list**。这个结构指向等待队列中第一个进程和最后一个进程。

10.1.4 节描述了等待队列实现的例子。通常, 进程让自己睡眠涉及对某个 `wait_event*` 宏的调用 (稍后讨论), 或者, 如同第 10 章所举的例子一样, 可以通过执行下列步骤来实现。

(1) 通过声明等待队列, 进程利用 `DECLARE_WAITQUEUE_HEAD` 继续睡眠。

(2) 利用 `add_wait_queue()` 或 `add_wait_queue_exclusive()` 把自己加入等待队列。

(3) 把进程状态变为 `TASK_INTERRUPTIBLE` 或 `TASK_UNINTERRUPTIBLE`。

(4) 检测外部事件, 如果外部事件还没有发生, 则调用 `schedule()`。

(5) 外部事件发生后, 把进程设置为 `TASK_RUNNING` 状态。

(6) 通过调用 `remove_wait_queue()` 从等待队列删除自己。

通过调用某个 `wake_up` 宏来唤醒进程。这些宏唤醒某个等待队列上的所有进程。它把进程置为 `TASK_RUNNING` 状态并放回运行队列。

让我们来看看调用函数 `add_wait_queue()` 时发生了什么。

3.7.1 添加到等待队列

有两个不同的函数可以向等待队列添加睡眠进程, 这就是函数 `add_wait_queue()` 和 `add_wait_queue_exclusive()`, 它们分别对应两种类型的睡眠进程。非独占式等待进程是所等待的条件返回时不被其他进程所共享, 而独占式等待进程则在等待一个其他进程可能正在等待的条件, 这就可能会产生一个竞态条件。

`add_wait_queue()` 函数向等待队列插入一个非独占式进程。非独占式进程是指在任何条件下, 当等待的事件完成时就被内核唤醒的进程。这个函数设置等待队列结构的 `flags` 字段, 表示把睡眠进程的标志的相应位设为 0, 同时设置等待队列锁, 以避免中断访问同一个等待队列而产生竞争条件, 之后把 `wait_queue_t` 结构加入等待队列链表, 并且从等待队列恢复锁, 使得其他进程可以使用。

```
-----
kernel/fork.c
93 void add_wait_queue(wait_queue_head_t *q, wait_queue_t * wait)
94 {
95     unsigned long flags;
96
97     wait->flags &= ~WQ_FLAG_EXCLUSIVE;
98     spin_lock_irqsave(&q->lock, flags);
99     __add_wait_queue(q, wait);
100     spin_unlock_irqrestore(&q->lock, flags);
101 }
```

`add_wait_queue_exclusive()` 函数向等待队列插入一个独占式进程。它把等待队列结构的 `flags` 字段的相应位设置为 1, 并且以与 `add_wait_queue()` 大致相同的独占方式进行操作, 但有一个例外, 它将独占进程添加到队列的末尾。这意味着在一个特定的等待队列里, 非独占式进程在前面, 独占式进程在后面。与讨论唤醒进程时所看到的一样, 这个次序也就是等待队列中的进程被唤醒的次序。

```
-----
kernel/fork.c
105 void add_wait_queue_exclusive(wait_queue_head_t *q, wait_queue_t * wait)
106 {
107     unsigned long flags;
108
109     wait->flags |= WQ_FLAG_EXCLUSIVE;
110     spin_lock_irqsave(&q->lock, flags);
111     __add_wait_queue_tail(q, wait);
112     spin_unlock_irqrestore(&q->lock, flags);
113 }
-----
```

3.7.2 等待事件

虽然在 Linux 2.6 内核中仍然支持 `sleep_on()`、`sleep_on_timeout()` 和 `interruptible_sleep_on()` 接口, 但是在 2.7 版本中将会去掉这几个接口。因此, 此处只介绍将要取代 `sleep_on*`() 接口的 `wait_event*`() 接口。

`wait_event*`() 接口包括 `wait_event()`、`wait_event_interruptible()` 和 `wait_event_interruptible_timeout()`。图 3-15 显示了这些函数的基本调用轨迹。

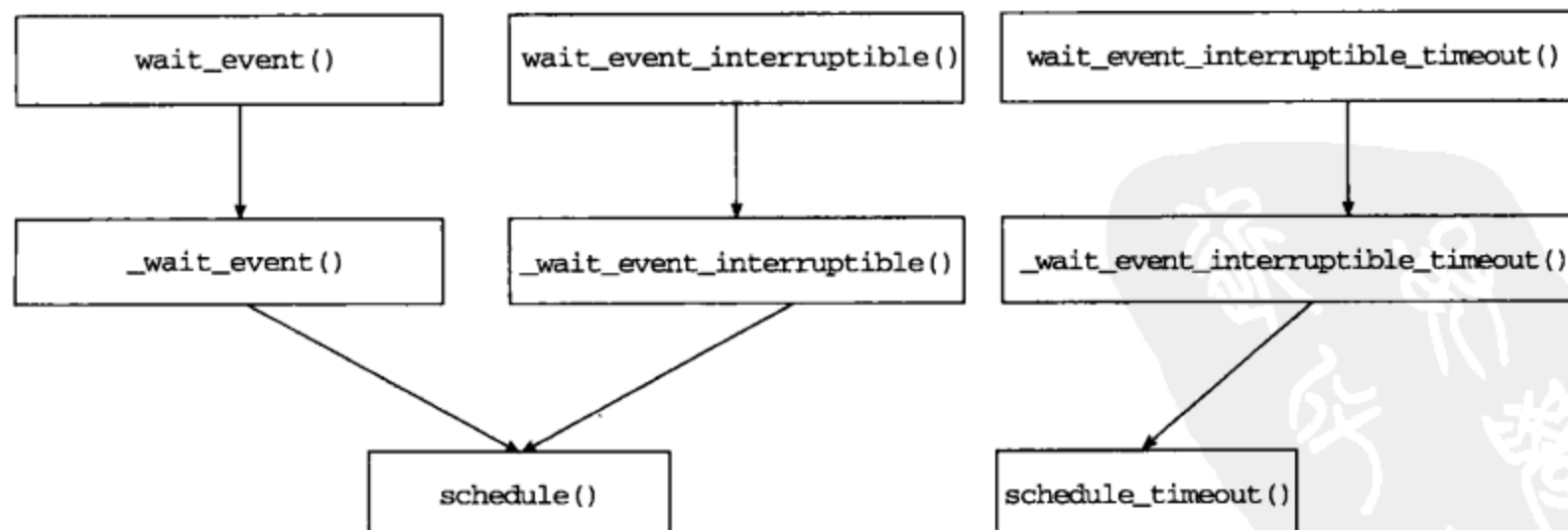


图 3-15 `wait_event*`() 的调用图

本节将详细分析并描述与 `wait_event()` 相关的接口, 同时也会提及相对于其他两个函数有什么差别。`wait_event()` 接口是对 `__wait_event()` 调用以无限循环的方式进行包装, 仅当所等待的条件返回时, 才退出循环。`wait_event_interruptible_timeout()` 传递的第三个参数是 `int` 类型的 `ret`, 用于传递超时时间。

`wait_event_interruptible()`是这三个接口中唯一有返回值的接口。如果一个信号中断了等待事件，这个返回值就是`-ERESTARTSYS`，否则，如果条件成立，就返回0。

```
-----
include/linux/wait.h
137 #define wait_event(wq, condition)
138 do {
139     if (condition)
140         break;
141     __wait_event(wq, condition);
142 } while (0)
-----
```

`__wait_event()`接口完成所有围绕进程状态改变和描述符操作的工作。

```
-----
include/linux/wait.h
121 #define __wait_event(wq, condition)
122 do {
123     wait_queue_t __wait;
124     init_waitqueue_entry(&__wait, current);
125
126     add_wait_queue(&wq, &__wait);
127     for (;;) {
128         set_current_state(TASK_UNINTERRUPTIBLE);
129         if (condition)
130             break;
131         schedule();
132     }
133     current->state = TASK_RUNNING;
134     remove_wait_queue(&wq, &__wait);
135 } while (0)
-----
```

第124~126行：为当前进程初始化等待队列描述符，并把该描述符添加到上层调用传递过来的等待队列中。到目前为止，`__wait_event_interruptible`和`__wait_event_interruptible_timeout`看起来与`__wait_event`完全相同。

第127~132行：这几行代码设置一个无限循环，仅当条件成立时才退出循环。进程在这个条件上阻塞前，利用`set_current_state`宏将其状态设置为`TASK_INTERRUPTIBLE`。回想一下，这个宏引用了指向当前进程的指针，因此不必把进程信息传递给它。该进程阻塞以后，它就通过调用`schedule()`将CPU让给其他进程。在这一点上，`__wait_event_interruptible()`与它有很大的不同；该函数把进程的状态字段设置成`TASK_UNINTERRUPTIBLE`，并让它等待`signal_pending`调用的返回，`signal_pending`调用用于等待发送给当前进程的信号。`__wait_event_interruptible_timeout`与`__wait_event_interruptible`非常类似，区别是：当调用调度程序时，前者调用`schedule_timeout()`，而不是`schedule()`。`schedule_timeout`把传递给原始接口`wait_event_interruptible_timeout`的超时长度参数作为自己的参数。

第133~134行：此处，条件已经满足，或者如果是在另外两个接口的情况下，就意味着进程可能收到一个信号或者已经超时。现在，将进程描述符的`state`字段设置回`TASK_RUNNING`（调

度程序把它放入运行队列)。最后，从等待队列中删除描述符`_wait`。`remove_wait_queue()`函数会在删除描述符前为等待队列上锁，而在返回以前再为它解锁。

3.7.3 唤醒进程

进程必须被唤醒以检查它等待的条件是否成立。注意，进程可以让自己睡眠，但是不能唤醒自己。除了有三个主要的“唤醒”函数以外，还有许多宏可以用来“唤醒”等待队列中的进程。宏`wake_up`、`wake_up_nr`、`wake_up_all`、`wake_up_interruptible`、`wake_up_interruptible_nr`和`wake_up_interruptible_all`都以不同的参数来调用`__wake_up()`。宏`wake_up_all_sync`和`wake_up_interruptible_sync`都以不同的参数来调用`__wake_up_sync()`。最后，`wake_up_locked`宏默认为`__wake_up_locked()`函数：

```
-----
include/linux/wait.h
116 extern void FASTCALL(__wake_up(wait_queue_head_t *q, unsigned int mode, int nr));
117 extern void FASTCALL(__wake_up_locked(wait_queue_head_t *q, unsigned int mode));
118 extern void FASTCALL(__wake_up_sync(wait_queue_head_t *q, unsigned int mode, int
    nr));
119
120 #define wake_up(x)    __wake_up((x), TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE, 1)
121 #define wake_up_nr(x, nr) __wake_up((x), TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE, nr)
122 #define wake_up_all(x) __wake_up((x), TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE, 0)
123 #define wake_up_all_sync(x) __wake_up_sync((x), TASK_UNINTERRUPTIBLE |
    TASK_INTERRUPTIBLE, 0)
124 #define wake_up_interruptible(x) __wake_up((x), TASK_INTERRUPTIBLE, 1)
125 #define wake_up_interruptible_nr(x, nr) __wake_up((x), TASK_INTERRUPTIBLE, nr)
126 #define wake_up_interruptible_all(x) __wake_up((x), TASK_INTERRUPTIBLE, 0)
127 #define wake_up_locked(x)    __wake_up_locked((x), TASK_UNINTERRUPTIBLE |
    TASK_INTERRUPTIBLE)
128 #define wake_up_interruptible_sync(x) __wake_up_sync((x), TASK_INTERRUPTIBLE, 1
129 )
-----
```

让我们来看看`__wake_up()`：

```
-----
kernel/sched.c
2336 void fastcall __wake_up(wait_queue_head_t *q, unsigned int mode, int
    nr_exclusive)
2337 {
2338     unsigned long flags;
2339
2340     spin_lock_irqsave(&q->lock, flags);
2341     __wake_up_common(q, mode, nr_exclusive, 0);
2342     spin_unlock_irqrestore(&q->lock, flags);
2343 }
-----
```

第 2336 行：传给`__wake_up`的参数包括：**q**，一个指向等待队列的指针；**mode**，表示将要唤醒的线程类型（由线程的状态来标识）；**nr_exclusive**，表示是独占式唤醒还是非独占式唤醒。

独占式唤醒 (`nr_exclusive = 0` 时) 将唤醒等待队列中的所有进程 (包括独占式进程和非独占式进程), 而非独占式唤醒只唤醒一个独占式进程和所有的非独占式进程。

第 2340 和 2342 行: 这两行分别设置和解除等待队列的自旋锁。在调用 `__wake_up_common()` 之前设置该锁, 以确保不会出现竞态条件。

第 2341 行: 函数 `__wake_up_common` 完成 `wakeup` 函数的大部分工作:

```
-----
kernel/sched.c
2313 static void __wake_up_common(wait_queue_head_t *q,
unsigned int mode, int nr_exclusive, int sync)
2314 {
2315     struct list_head *tmp, *next;
2316
2317     list_for_each_safe(tmp, next, &q->task_list) {
2318         wait_queue_t *curr;
2319         unsigned flags;
2320         curr = list_entry(tmp, wait_queue_t, task_list);
2321         flags = curr->flags;
2322         if (curr->func(curr, mode, sync) &&
2323             (flags & WQ_FLAG_EXCLUSIVE) &&
2324             !--nr_exclusive)
2325             break;
2326     }
2327 }
```

第 2313 行: 传给 `__wake_up_common` 的参数分别是: **q**, 一个指向等待队列的指针; **mode**, 将要唤醒的线程类型; **nr_exclusive**, 如前所示的唤醒类型; **sync**, 表示唤醒是否应该同步。

第 2315 行: 此处为链表元素操作设置临时指针。

第 2317 行: `list_for_each_safe` 宏扫描等待队列的每一项, 这是循环的开始。

第 2320 行: `list_entry` 宏返回 `tmp` 变量所存放的等待队列结构的地址。

第 2322 行: 调用 `wait_queue_t` 的 `func` 字段。默认情况下, 它调用如下所示的 `default_wake_function()`:

```
-----
kernel/sched.c
2296 int default_wake_function(wait_queue_t *curr, unsigned mode,
int sync)
2297 {
2298     task_t *p = curr->task;
2299     return try_to_wake_up(p, mode, sync);
2300 }
```

这个函数对 `wait_queue_t` 结构所指向的进程调用 `try_to_wake_up()` (参见 `kernel/sched.c`)。它完成唤醒进程的大部分工作, 包括把进程放入运行队列。

第 2322~2325 行: 如果被唤醒的进程是第一个独占式进程, 则终止循环。如果知道所有独占式进程都排在等待队列末尾的话, 这将是很有意义的。当在等待队列中遇到第一个独占式进程后,

剩余的所有进程也将是独占式的，因此，不应该唤醒它们，而应跳出循环。

3.8 异步执行流程

前面已经介绍过，进程能够通过中断从一个状态转换到另一个状态，例如从 `TASK_INTERRUPTIBLE` 到 `TASK_RUNNING`。获得这种转换的途径之一是包括异常和中断在内的异步执行。同时，进程也可以在用户态和内核态之间来回切换。接下来首先介绍异常是如何工作的，之后，再来分析中断究竟是如何工作的。

3.8.1 异常

异常 (exception)，也称作**同步中断** (synchronous interrupts)，是完全发生在处理器硬件内部的事件。这些事件和处理器的执行是同步的，也就是说，它们不是发生在代码指令执行期间，而是发生在代码指令执行之后。处理器异常的例子包括对一个没有映射到物理内存的虚拟存储单元的引用 (称为**缺页**)，以及导致被 0 除的计算。要强调的一点是，异常 (有时也称为软中断 `irq`) 通常发生在指令执行之后，这使得它们与外部事件或**异步事件**有很大的区别，稍后将会在 3.8.2 节中详细讨论。

大多数现代处理器 (包括 x86 和 PPC) 允许程序员执行某些指令来产生一个异常。这些指令可以被看做是用来辅助硬件的子例程调用，例如**系统调用**。

系统调用

Linux 为用户态的程序提供了进入内核的入口点，可以通过这个入口点从内核请求服务或访问硬件。这些入口点在内核中是经过标准化的、预先定义好的。用户态的程序可用的许多 C 库例程，例如图 3-9 中的 `fork()` 函数，把代码和一个或多个系统调用捆绑在一起形成一个单独的函数。当用户进程调用其中某个函数时，相应的值被放入适当的处理器寄存器中，并产生一个软中断。然后，再由这个软中断来调用内核入口点。系统调用 (syscalls) 也可以由内核代码来访问，尽管不推荐这样做。从何处访问系统调用是我们讨论的源点，因为从内核调用系统调用可以提高系统性能。这种性能上的提高要和增加的复杂性以及代码的可维护性进行权衡。本节讨论“传统的”从用户空间调用系统调用的实现过程。

系统调用能够在用户空间和内核空间之间传送数据。内核提供了两个函数来完成这种数据传送：`copy_to_user()` 和 `copy_from_user()`。和所有内核编程一样，传送数据时，指针、长度、描述符和许可权的有效性是很关键的。这些函数都内置了有效性的检验。有趣的是，它们返回的是未被传送的字节数。

系统调用自身的特点决定了它的实现是与硬件相关的。在传统的 Intel 体系结构中，所有的系统调用都使用软中断 `0x80`^①。

将系统调用的参数传入通用寄存器，其唯一的系统调用号被放入 `%eax`。系统调用在 x86 体系

① 新的 (PIV+) Intel 处理器在提高性能方面做了许多努力，其中大部分工作随着 `vsyscalls` 的实现而得以完成。`vsyscalls` 基于对用户空间内存 (特别是“`vsyscall`”页) 的访问，它使用了比传统的 `int 0x80` 调用更快的 `sysenter` 和 `sysexit` 指令 (当可用时)。在许多 PPC 的实现中也追求类似的性能提高。

结构中的实现把参数个数限制为 5 个。如果函数的参数多于 5 个，可以给函数传递一个指向参数块的指针。在执行汇编指令 `int 0x80` 时，一个特殊的内核态例程将借助处理器的异常处理能力被调用执行。下面来看一个如何初始化系统调用入口的例子：

```
set_system_gate(SYSCALL_VECTOR, &system_call);
```

这个宏在入口 128 (`SYSCALL_VECTOR`) 处创建一个用户特权描述符，指向 `entry.S` 中系统调用处理程序 (`system_call`) 的地址。

和我们在下一节中将看到的一样，PPC 中断例程被“固定”到某些存储单元；外部中断处理程序被固定到地址 `0x500` 处，系统定时器被固定到地址 `0x900` 处，等等。系统调用指令 `sc` 指向地址 `0xc00`。接下来分析一下 `head.S` 中的代码片断，这里的处理程序是为 PPC 系统调用设置的。

```
-----
arch/ppc/kernel/head.S
484  /* System call */
485  . = 0xc00
486  SystemCall:
487  EXCEPTION_PROLOG
488  EXC_XFER_EE_LITE(0xc00, DoSyscall)
-----
```

第 485 行：地址的定位。这一行告诉装载程序下一条指令将被定位在地址 `0xc00` 处。由于标号遵循类似的规则，因此，标号 `SystemCall` 连同宏 `EXCEPTION_PROLOG` 的第一行代码都从地址 `0xc00` 处开始。

第 488 行：这个宏分派 `DoSyscall()` 处理程序。

在 x86 和 PPC 这两种体系结构中，系统调用号和所有的参数都是存入处理器的寄存器中。

当 x86 的异常处理程序处理 `int0x80` 时，将对系统调用表进行索引。文件 `arch/i386/kernel/entry.S` 中包含了低级中断处理例程和系统调用表 `sys_call_table`。同样，对于 PPC 而言，系统调用的低级中断处理例程是在 `arch/ppc/kernel/entry.S` 中定义的，而 `sys_call_table` 则在 `arch/ppc/kernel/misc.S` 中。

系统调用表是用汇编代码实现的 C 数组，每个表项占 4 字节，每项被初始化为一个函数的地址。按照惯例，必须用“`sys_`”预先定义好函数名，因为表中的位置决定了系统调用号，因此必须把函数名称添加到表的尾部。即使使用不同的汇编语言，不同体系结构的系统调用表也几乎是完全一样的。然而，直到本书撰写之时，PPC 的系统调用表还只有 255 个表项，而 x86 的系统调用表有 275 个表项。

文件 `include/asm-i386/unistd.h` 和 `include/asm-ppc/unistd.h` 把系统调用和它在系统调用表中的位置号相关联。在该文件中，“`sys_`”被替换成“`__NR_`”，同时还包含辅助用户程序把参数装入寄存器的宏（2.3 节介绍了 C 和汇编变量以及内嵌汇编程序）。

接下来看一下如何添加一个名为 `sys_ourcall` 的系统调用。系统调用必须被添加到 `sys_call_table`。下列代码就是把系统调用加入 x86 的 `sys_call_table`：

```

-----
arch/i386/kernel/entry.S
607  .data
608  ENTRY(sys_call_table)
609  .long sys_restart_syscall /* 0 - old "setup()" system call, used for restarting*/
...
878  .long sys_tgkill /* 270 */
879  .long sys_utimes
880  .long sys_fadvise64_64
881  .long sys_ni_syscall /* sys_vserver */
882  .long sys_ourcall /* our syscall will be 274 */
883
884  nr_syscalls=(.-sys_call_table)/4
-----

```

在 x86 体系结构中，该系统调用编号为 274。若要在 PPC 中添加名为 sys_ourcall 的系统调用，其入口将是第 255 号。此处，当把系统调用及其位置号的关联引入文件 include/asm-ppc/unistd.h 时，会看到它是什么样子的。__NR_ourcall 就是入口号 255，位于系统调用表的表尾。

```

-----
include/asm-ppc/unistd.h
/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
...
#define __NR_utimes 271
#define __NR_fadvise64_64 272
#define __NR_vserver 273
#define __NR_ourcall 274

/* #define NR_syscalls 274 this is the old value before our syscall */
#define NR_syscalls 275
-----

```

下一节将讨论中断和通知内核需要处理中断时所涉及的硬件。作为与中断稍有差异的异常，其处理程序将会在适当的时候被调用。尽管异常和中断在处理时采用的方法相同，但异常趋向于给当前进程发回信号，而不是作用于硬件设备。

3.8.2 中断

中断和处理器的执行是异步的，这意味着中断发生在两条指令之间。处理器通过传到其引脚（INTR 或者 NMI）的外部信号来接收中断通知。这个信号来自于称为中断控制器的硬件设备。中断和中断处理器都是与硬件及系统相关的。不同体系结构对中断控制器的设计有很大不同。本节简要介绍独立于体系结构的部分和依赖于体系结构的部分在硬件方面的主要差异，并简单分析用于追踪内核代码的函数。

由于在任何给定的时刻，处理器必须与若干外部设备中的某一个进行通信，因而中断控制器是必不可少的。早期的 x86 计算机使用了一对级联的 Intel 8259 中断处理器^①，这些中断处理器通过配置，能够辨别 15 根离散的中断线（IRQ）（参见图 3-16）。当中断控制器有一个待处理的中断（例如，当你按下一个键）时，它将触发连接到处理器上的 INTA 线。然后，处理器通过激活连接到中断控制器 INTA 线上的应答线来应答这个信号。此时，中断控制器才把 IRQ 数据传送到处理器。以上过程被称为一个中断应答周期。

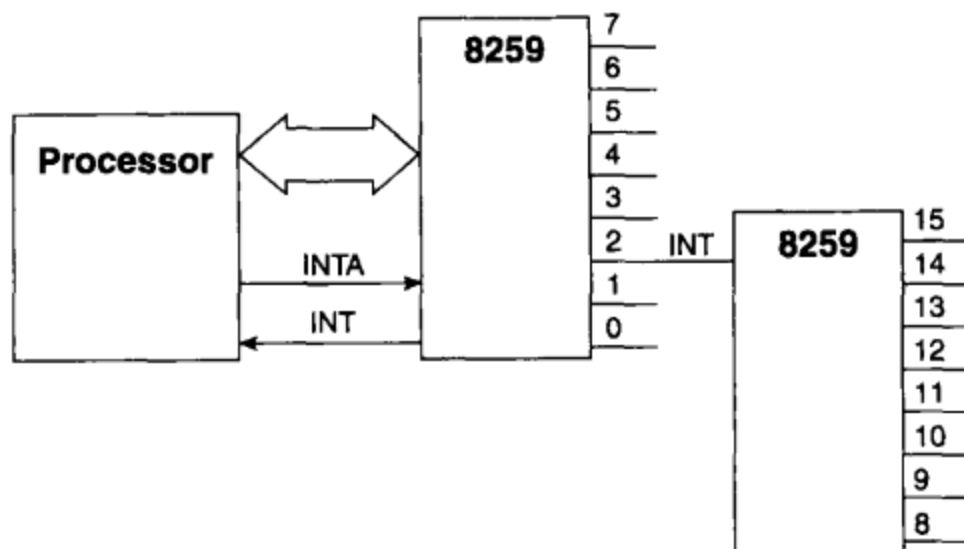


图 3-16 级联的中断控制器

更新的 x86 处理器拥有一个本地的 APIC（Advanced Programmable Interrupt Controller，高级可编程中断控制器）。本地 APIC（被嵌入在处理器包中）接收来自以下中断源的中断信号：

- ☐ 处理器的中断引脚（LINT0, LINT1）；
- ☐ 内部定时器；
- ☐ 内部性能监控器；
- ☐ 内部温度传感器；
- ☐ 内部 APIC 错误；
- ☐ 另外一个处理器（用于处理器间的中断）；
- ☐ 外部 I/O APIC（经由多处理器系统上的 APIC 总线）。

APIC 收到一个中断信号后，就把这个信号交给处理器核心（在处理器包中）。图 3-17 中显示的 I/O APIC 是处理器芯片集的一部分，它是为接收 24 个可编程的中断输入而设计的。

对于具有本地 APIC 的 x86 处理器，你也可以将其（中断模式）配置成传统的 8259 类型的中断控制器，而不使用 I/O APIC 体系架构（或者说 I/O APIC 可以被配置成 8259 控制器的接口，以 8259 的方式使用）。为了查明一个系统是否在使用 I/O APIC 体系结构，可在命令行输入如下命令：

```
lkp:~# cat /proc/interrupts
```

如果看到了列出的 I/O-APIC，说明 I/O APIC 正在使用，如果看到的是 XT-PIC，就意味着正在使用 8259 类型的体系结构。

^① 第二个 8259 的输出连接到第一个 8259 的 IRQ（通常为 IRQ2）上。

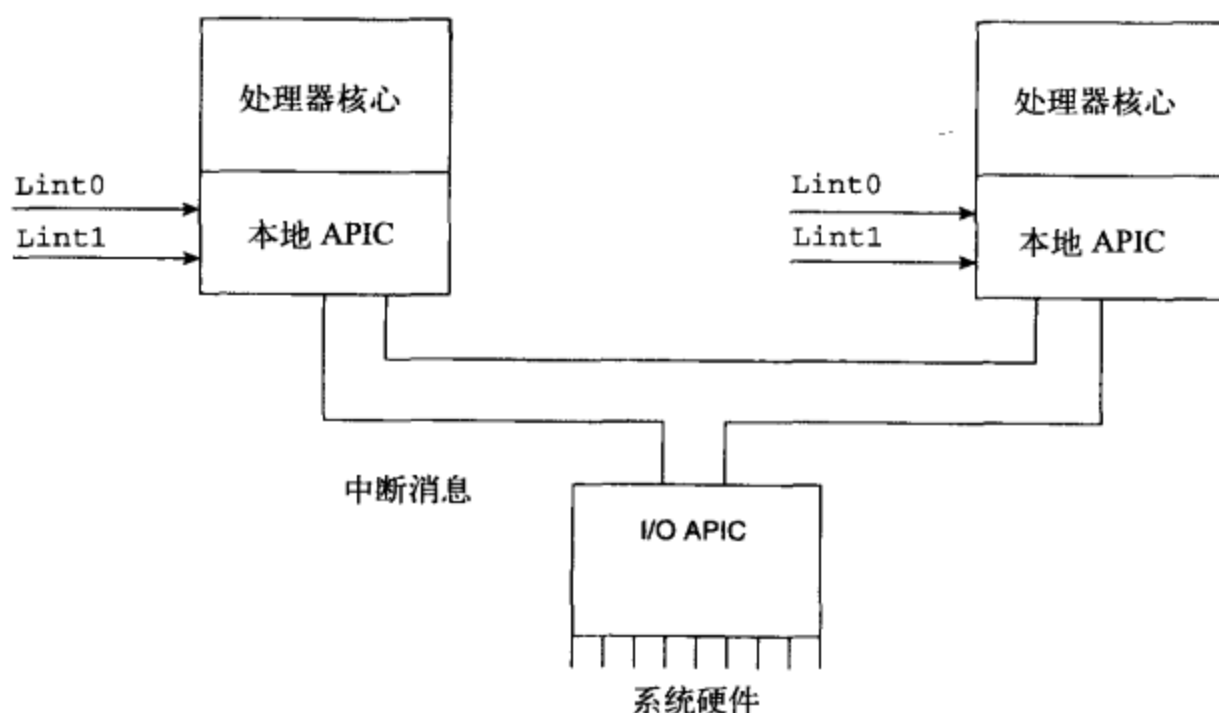


图 3-17 I/O APIC

Power Mac G4 和 G5 的 PowerPC 中断控制器被集成到 Key Largo 和 K2 I/O 控制器中。可在命令行输入这样的命令：

```
lkp:~# cat /proc/interrupts
```

在 G4 机上产生了 OpenPIC，它是一个由 AMD 和 Cyrix 于 1995 年为多处理器系统发起的开放可编程中断控制器（Open Programmable Interrupt Controller, OpenPIC）标准。MPIC 是 IBM 对 OpenPIC 的实现，并用在他们的几个公共硬件平台参数（CHRP）设计中。以前的苹果机有一个用于 4xx 嵌入式处理器的内部中断控制器，中断控制器的核心被集成到 ASIC 芯片中。

既然已经讨论过硬件为什么、如何以及何时给内核传送中断请求，接下来就可以分析内核处理硬件系统时钟中断的这个实际例子，并详细叙述中断将在何处被传送。在分析系统定时器的代码时，可以看到，在 x86 和 PPC 体系结构中中断发生时，硬件到软件的接口都是用跳转表来实现的，跳转表用来为给定的中断选择相应的处理程序。

x86 体系结构的每个中断都被赋予一个唯一的编号或者向量。中断发生时，就用这个向量来索引 IDT（Interrupt Descriptor Table，中断描述符表），可参阅 x86 门描述符格式的《Intel 程序员参考资料》。IDT 允许硬件在中断时辅助软件进行地址解析，并检查处理程序代码的特权。PPC 体系结构中的处理稍有不同，它使用编译时创建的中断向量表来执行适当的中断处理程序。（稍后在比较 x86 和 PPC 的系统定时器的中断处理过程时，将会看到这两种体系结构在软件的初始化和跳转表的使用上有更多的不同。）下一节讨论中断处理程序及其实现。接下来，作为 Linux 中断及其相关处理程序实现的一个例子，将讨论系统定时器。

首先来看看不同种类的中断处理程序。

1. 中断处理程序

中断和异常处理程序很像常规的 C 函数。它们可能并经常调用硬件特有的汇编例程。Linux 中断处理程序划分为高性能的上半部分和低性能的下半部分。

□ 上半部分 (top half)。这一部分必须尽快执行。上半部分处理程序可以在所有本地中断（对

给定处理器来说)禁用的情况下运行(也叫快速处理程序),这依赖于它们如何被注册。上半部分中的代码必须仅限于直接响应硬件或者执行紧迫的进程。在上半部分中拖延时间会大大影响系统的性能。为了保持高性能和低延迟(响应设备所花的时间),引入了下半部分体系结构。

- 下半部分(bottom half)。这一部分允许处理程序编写者把不太紧急的工作推后,直到内核有充裕的时间来处理^①。请注意,中断随着系统的执行异步地到达;此刻,内核也许正在做更紧迫的事情。有了下半部分体系结构,中断处理程序编写者可以让内核稍迟一点运行不太紧急的处理程序代码。

表3-8解释了下半部分中断处理的4个最普通的方法。

表3-8 下半部分中断处理的方法

“陈旧的”下半部分	无论处理器的个数有多少,每次只有一个下半部分能够运行,因此,这些每个SMP上都有下半部分处理程序被逐步淘汰。这种系统已经在2.6内核中删除,只在参考时提及
工作队列	据说上半部分代码运行在中断上下文中,这意味着它不和任何进程关联。由于没有进程关联,代码就不能睡眠或阻塞。工作队列运行在进程上下文中,并具有任何内核线程具备的能力。它有一个丰富的函数集,用于创建、调度、撤销进程,等等。有关工作队列的更多信息,参见10.1.5节
软中断	软中断也运行在中断上下文中,它类似于下半部分,但有一个例外,那就是同一类型的软中断能够同时运行在多个处理器上。系统中只有32个软中断。系统定时器就使用软中断
tasklet	类似于软中断,但不存在限制。所有的tasklet都通过一个软中断来产生,同一个tasklet不能在多个处理器上同时运行。和软中断相比,tasklet的接口更容易实现和使用

2. IRQ结构

有三个主要的结构包含了与IRQ相关的所有信息:irq_desc_t、irqaction和hw_interrupt_type。图3-18解释了它们之间是如何相互关联的。

● irq_desc_t结构

irq_desc_t结构是主要的IRQ描述符,它存放在大小为NR_IRQS(它的值依赖于体系结构)的全局数组irq_desc中。

```
-----
include/linux/irq.h
60 typedef struct irq_desc {
61     unsigned int status; /* IRQ 状态 */
62     hw_irq_controller *handler;
63     struct irqaction *action; /* IRQ 操作链表 */
64     unsigned int depth; /* 禁止嵌套的 irq */
65     unsigned int irq_count; /* 用于检查所打断的中断 */
66     unsigned int irqs_unhandled;
67     spinlock_t lock;
68 } ____cacheline_aligned irq_desc_t;
69
70 extern irq_desc_t irq_desc [NR_IRQS];
-----
```

① Linux的早期版本为系统定时器使用了上半部分/下半部分处理程序。后来,对此进行了重写,只保留了高性能的上半部分。

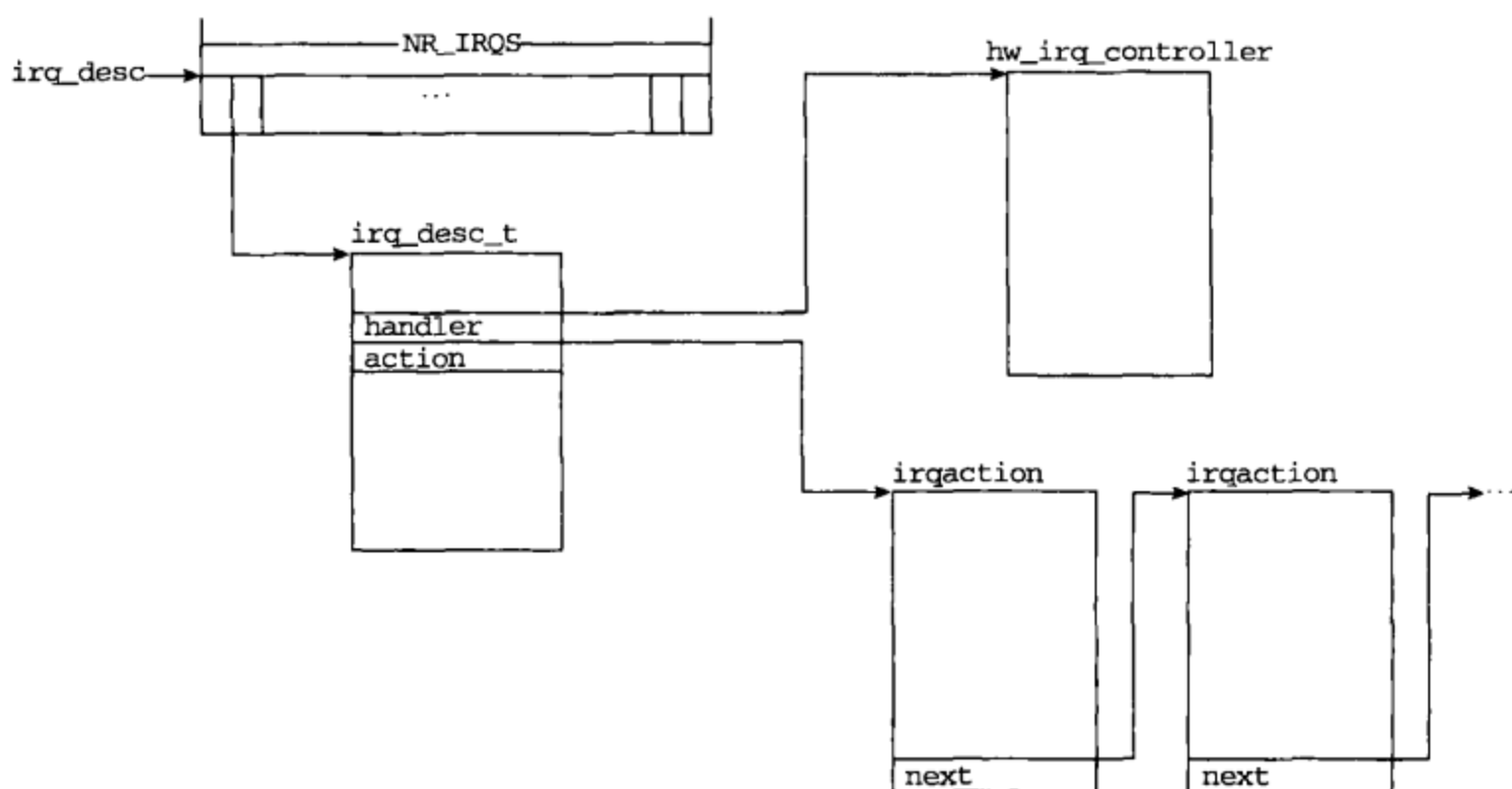


图 3-18 IRQ 结构

第 61 行：设置标志来描述 IRQ 线的状态，从而确定 status 字段的值。表 3-9 给出了这些标志。

表3-9 irq_desc_t->field标志

标 志	说 明
IRQ_INPROGRESS	表示正在为IRQ线执行处理程序
IRQ_DISABLED	表示IRQ被软件禁止，因此，即使启用了物理请求线本身，它的处理程序也不会被执行
IRQ_PENDING	中间状态，表示中断的发生已被确认，但是处理程序还没有执行
IRQ_REPLAY	前一个IRQ还没有被确认
IRQ_AUTODETECT	当IRQ线被探测时所设置的状态
IRQ_WAITING	探测时使用
IRQ_LEVEL	IRQ是电平触发，而不是边沿触发的
IRQ_MASKED	这个标志在内核代码中不使用
IRQ_PER_CPU	用于表明IRQ线对CPU调用是局部的

第 62 行：handler 字段是指向 hw_irq_controller 的一个指针。hw_irq_controller 是对 hw_interrupt_type 结构的类型定义，是用来描述低层硬件的中断控制器描述符。

第 63 行：action 字段包含一个指向 irqaction 结构的指针。当 IRQ 被启用时，记录中断处理程序例程的这个结构将被执行，稍后将详细介绍这个结构。

第 64 行：depth 字段是禁止 IRQ 的嵌套计数器。只有当这个字段的值为 0 时，IRQ_DISABLE 才被清零。

第 65~66 行：irq_count 字段和 irqs_unhandled 字段共同确定可能被延迟的中断请求。它们在 x86 和 PPC64 的函数 note_interrupt() (arch/<arch>/kernel/irq.c) 中使用。

第 67 行：lock 字段存放描述符的自旋锁。

● irqaction 结构

内核使用 irqaction 结构来记录中断处理程序及其与 IRQ 的关联。接下来看看下一节将要讨论的它的结构和字段：

```
-----
include/linux/interrupt.h
35 struct irqaction {
36     irqreturn_t (*handler) (int, void *, struct pt_regs *);
37     unsigned long flags;
38     unsigned long mask;
39     const char *name;
40     void *dev_id;
41     struct irqaction *next;
42 };
-----
```

第 36 行：**handler** 字段是指向中断处理程序的指针，当发生相应的中断时就调用这个处理程序。

第 37 行：flags 字段包含了 SA_INTERRUPT 之类的标志，表示中断处理程序将在所有中断禁用的情况下运行，SA_SHIRQ 表示处理程序可以和其他处理程序共享一根 IRQ 线。

第 39 行：name 字段用于存放被注册的中断名称。

● hw_interrupt_type 结构

hw_interrupt_type 或 hw_irq_controller 包含了与系统的中断控制器相关的所有数据。首先来先分析这个结构，然后再讨论如何为一对中断控制器实现该结构。

```
-----
include/linux/irq.h
40 struct hw_interrupt_type {
41     const char * typename;
42     unsigned int (*startup)(unsigned int irq);
43     void (*shutdown)(unsigned int irq);
44     void (*enable)(unsigned int irq);
45     void (*disable)(unsigned int irq);
46     void (*ack)(unsigned int irq);
47     void (*end)(unsigned int irq);
48     void (*set_affinity)(unsigned int irq, cpumask_t dest);
49 };
-----
```

第 41 行：typename 用于存放 PIC (Programmable Interrupt Controller, 可编程中断控制器) 的名称 (PIC 将在后面详细介绍)。

第 42~48 行：这些字段用于存放一些指向 PIC 特有的编程函数的指针。

现在，来看看 PPC 的中断控制器吧。首先看看 PowerMac 的 PIC：

```
-----
arch/ppc/platforms/pmac_pic.c
170 struct hw_interrupt_type pmac_pic = {
```

```

171  " PMAC-PIC ",
172  NULL,
173  NULL,
174  pmac_unmask_irq,
175  pmac_mask_irq,
176  pmac_mask_and_ack_irq,
177  pmac_end_irq,
178  NULL
179  };

```

可以看到，这个 PIC 的名字是 PMAC-PIC，它定义了 6 个函数中的 4 个。其中，`pmac_unmask_irq` 和 `pmac_mask_irq` 函数分别用于启用和禁用 IRQ 线，函数 `pmac_mask_and_ack_irq` 用于确认 IRQ 已收到，函数 `pmac_end_irq` 则负责中断处理程序执行完后的清理工作。

```

-----
arch/i386/kernel/i8259.c
59  static struct hw_interrupt_type i8259A_irq_type = {
60  "XT-PIC",
61  startup_8259A_irq,
62  shutdown_8259A_irq,
63  enable_8259A_irq,
64  disable_8259A_irq,
65  mask_and_ack_8259A,
66  end_8259A_irq,
67  NULL
68  };

```

x86 的 8259 PIC 被称为 XT-PIC，它定义了最前面的 5 个函数。前两个函数 `startup_8259A_irq` 和 `shutdown_8259A_irq`，分别用于启动和关闭实际的 IRQ 线。

3. 中断示例：系统定时器

系统定时器是操作系统的脉搏。系统启动时，系统定时器及其中断在系统初始化期间被初始化。此时对中断进行初始化所使用的接口与运行时中断被注册所使用的接口不同。接下来分析这个示例时将指出这两者的差异。

因为有更复杂的芯片生产出来，内核设计者对于系统定时器的来源多了几种选择。对 x86 体系结构来说，最常见的定时器实现是 PIT (Programmable Interval Timer, 可编程间隔定时器)，对 PowerPC 来说，最常见的定时器是衰减器。

x86 体系结构曾经用 Intel 8254 定时器实现 PIT。8254 用做 16 位的递减计数器——终端计数中断，即在寄存器中写入一个数值，8254 递减这个值，直到它变成 0。此时，它将激活 8259 中断控制器上的 IRQ 0 输中断，IRQ 0 在本节前面已经提到过。

在 PowerPC 体系结构中实现的系统定时器是衰减定时器，它是一个 32 位的递减计数器，且以与 CPU 相同的频率运行。和 8259 类似，它将在达到其终止计数时激活一个中断。和 Intel 体系结构不同的是，衰减器被嵌入处理器中。

每当系统定时器递减，并激活一个中断，就称为一个节拍 (tick)。这个节拍的速率或频率由 **HZ** 变量设置。

HZ

HZ 是 Hertz 缩写 (Hz) 的变种, 因 Heinrich Hertz (1857—1894) 而命名。作为无线电波的创始人之一, 赫兹通过 Y 线环感应一个电火花证明了麦克斯韦关于电磁学的理论。然后, 马可尼在这些实验的基础上创立了现代无线电。为了向赫兹和他的成果表示敬意, 频率的基本单位就以他的名字来命名; 每秒 1 个周期等于 1 赫兹。

HZ 是在 `include/asm-xxx/param.h` 中定义的。让我们来看看这些值在 x86 和 PPC 中分别是多少。

```
-----
include/asm-i386/param.h
005  #ifdef __KERNEL__
006  #define HZ          1000      /* 内核内部定时器频率 */
kernel timer frequenay */
-----
include/asm-ppc/param.h
008  #ifdef __KERNEL__
009  #define HZ          100      /* 内核内部定时器频率 */
kernel timer frequenay */
-----
```

在大多数体系结构中, HZ 的值通常是 100, 但是因为机器的运行速率变得越来越快, 节拍率以某种模式增大了。本书中给出的两种主要体系结构的默认节拍速率都是 1 000。1 个节拍的周期是 $1/\text{HZ}$ 。这样, 周期 (或中断的间隔时间) 就是 1 毫秒。当 HZ 的值增大时, 将会在给定的时间里获得更多中断。尽管从计时功能来看这是更优的解决方案, 但是要注意重要的一点: 更多的处理器时间花费在应答内核的系统时钟中断上了。极端地说, 这可能减慢系统对用户态程序的响应。由于需要对所有的中断进行处理, 因此, 在两者之间找到恰当的平衡点是至关重要的。

现在开始分析系统定时器及其相关中断的初始化代码。系统定时器的处理程序是在内核初始化临近结束的时候才安装的。首先来看看代码段中的 `start_kernel()`, 这是系统引导时执行的主要初始化函数, 它首先调用 `trap_init()`, 接着调用 `init_IRQ()`, 最后再调用 `time_init()`:

```
-----
init/main.c
386  asmlinkage void __init start_kernel(void)
387  {
...
413  trap_init();
...
415  init_IRQ();
...
419  time_init();
...
}
```

第 413 行: 对于运行在保护模式下的 x86 体系结构, 宏 `trap_init()` 在中断描述符表 (IDT) 中初始化异常的入口。IDT 是在内存中设立的一张表。它的地址是在处理器的 IDTR 寄存器中设

置的。它的每个元素都是三种门当中的某一种。门是 x86 保护模式下的地址，由选择子、偏移量和特权级组成，其用途是转移程序的控制权。IDT 中有三种门，即系统门、中断门和陷阱门；所谓系统门就是把控制权转移给另一个进程；所谓中断门，就是在中断禁用的情况下，把控制权传递给中断处理程序；所谓陷阱门，就是在中断不改变的情况下，把控制权传递给中断处理程序。

PPC 被设计成跳转到一个特定地址这个地址依赖于异常。函数 `trap_init()` 对 PPC 来说是一个空操作。稍后继续分析系统定时器的代码时，将把 PPC 的中断描述符表和下面将要初始化的 x86 中断描述符表进行对比。

```
-----
arch/i386/kernel/traps.c
900 void __init trap_init(void)
901 {
902     #ifdef CONFIG_EISA
903         if (isa_readl(0x0FFFD9) == 'E' + ('I' << 8) + ('S' << 16) + ('A' << 24)) {
904             EISA_bus = 1;
905         }
906     #endif
907
908     #ifdef CONFIG_X86_LOCAL_APIC
909         init_apic_mappings();
910     #endif
911
912     set_trap_gate(0, &divide_error);
913     set_intr_gate(1, &debug);
914     set_intr_gate(2, &nmi);
915     set_system_gate(3, &int3); /* int3-5 can be called from all */
916     set_system_gate(4, &overflow);
917     set_system_gate(5, &bounds);
918     set_trap_gate(6, &invalid_op);
919     set_trap_gate(7, &device_not_available);
920     set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
921     set_trap_gate(9, &coprocessor_segment_overrun);
922     set_trap_gate(10, &invalid_TSS);
923     set_trap_gate(11, &segment_not_present);
924     set_trap_gate(12, &stack_segment);
925     set_trap_gate(13, &general_protection);
926     set_intr_gate(14, &page_fault);
927     set_trap_gate(15, &spurious_interrupt_bug);
928     set_trap_gate(16, &coprocessor_error);
929     set_trap_gate(17, &alignment_check);
930     #ifdef CONFIG_X86_MCE
931         set_trap_gate(18, &machine_check);
932     #endif
933     set_trap_gate(19, &simd_coprocessor_error);
934
935     set_system_gate(SYSCALL_VECTOR, &system_call);
936
937     /*
938     * default LDT is a single-entry callgate to lcall7 for iBCS
939     * and a callgate to lcall27 for Solaris/x86 binaries
940     */
941     set_call_gate(&default_ldt[0], lcall7);
942     set_call_gate(&default_ldt[4], lcall27);
943
```



```

944  /*
945  * Should be a barrier for any external CPU state.
946  */
947  cpu_init();
948
949  trap_init_hook();
950  }

```

第 902 行：查找 EISA 签名。isa_readl() 是一个辅助程序，它通过用 ioremap() 映射 I/O 来读取 EISA 总线。

第 908~910 行：如果存在一个高级可编程中断控制器 (APIC) 的话，就把它的地址添加到系统固定的地址映射表。对于“特殊的”系统地址辅助例程 set_fixmap_nocache()，请参阅文件 include/asm-i386/fixmap.h。init_apic_mappings() 使用这个例程来设置 APIC 的物理地址。

第 912~935 行：用陷阱门、系统门和中断门来初始化 IDT。

第 941~942 行：这些特殊的段间调用门来支持 Intel 的二进制兼容标准 (Intel Binary Compatibility Standard)，这个标准允许在 Linux 上运行其他的 UNIX 二进制程序。

第 947 行：初始化当前正在执行的 CPU 的中断描述符表和寄存器。

第 949 行：用于初始化系统特定的硬件，例如不同种类的 APIC。这对于大多数 x86 平台而言是一个空操作。

第 415 行^①：调用 init_IRQ() 来初始化硬件中断控制器。x86 和 PPC 体系结构都拥有多个设备的实现，对于 x86 体系结构，我们分析 i8259 设备。对于 PPC，分析和 Power Mac 相关的代码。

PPC 中 init_IRQ() 是在 arch/ppc/kernel/irq.c 中实现的。该函数根据特殊的硬件配置调用对应的某个例程来初始化 PIC。对于 Power Mac 配置而言，调用 arch/ppc/platforms/pmac_pic.c 中的函数 pmac_pic_init() 来初始化 G3、G4 和 G5 的 I/O 控制器。这是一个硬件特有的例程，它将设法识别 I/O 控制器的类型并对其进行适当的配置。在本例中，PIC 是 I/O 中断控制器设备的一部分，其中断初始化过程与 x86 类似，微小的区别在于，系统定时器不是在 PPC 的 init_IRQ() 函数而是在 time_init() 函数中启动的，本节稍后将介绍 time_init() 函数。

x86 体系结构对于 PIC 的选择比较少。如前所述，早期的系统使用级联的 8259，后来则使用 IOAPIC 体系结构。下面这段代码分析带有模拟 8259 中断控制器的 APIC：

```

-----
arch/i386/kernel/i8259.c
342 void __init init_ISA_irqs (void)
343 {
344     int i;
345     #ifdef CONFIG_X86_LOCAL_APIC
346     init_bsp_APIC();
347     #endif
348     init_8259A(0);
349     ...
351     for (i = 0; i < NR_IRQS; i++) {
352         irq_desc[i].status = IRQ_DISABLED;

```

① 原文中没有给出这一行代码，在内核源码中查找后，发现是位于 /init/main.c 中的 start_kernel() 调用了这个函数。

```

353     irq_desc[i].action = 0;
354     irq_desc[i].depth = 1;
355
356     if (i < 16) {
357         /*
358          * 16 old-style INTA-cycle interrupts:
359          */
360         irq_desc[i].handler = &i8259A_irq_type;
361     } else {
362         /*
363          * 'high' PCI IRQs filled in on demand
364          */
365         irq_desc[i].handler = &no_irq_type;
366     }
367 }
368 }
...
409
410 void __init init_IRQ(void)
411 {
412     int i;
413
414     /* all the set up before the call gates are initialized */
415     pre_intr_init_hook();
...
422     for (i = 0; i < NR_IRQS; i++) {
423         int vector = FIRST_EXTERNAL_VECTOR + i;
424         if (vector != SYSCALL_VECTOR)
425             set_intr_gate(vector, interrupt[i]);
426     }
...
431     intr_init_hook();
...
437     setup_timer();
...
}
-----

```

第 410 行：这是从 `start_kernel()` 调用的函数入口点，而 `start_kernel()` 是在系统启动时调用的主要的内核初始化函数。

第 342~348 行：如果本地 APIC 是可用的，并的确需要使用，那就将它初始化并设置为虚导线模式，用来和 8259 配合工作。然后，利用 `init_8259A(0)` 中的 I/O 寄存器来初始化 8259 设备。

第 422~426 行：在第 424 行，这个循环没有对中断向量 `syscalls` 进行设置，因为它早在 `tRap_init()` 中已经设置过了。Linux 使用 Intel 的中断门（已由内核代码初始化）作为中断描述符。这是由 `set_intr_gate()` 宏（在第 425 行）来设置的。异常则使用 Intel 的系统门和陷阱门，分别由 `set_system_gate()` 和 `set_trap_gate()` 来设置。这些宏是在文件 `arch/i386/kernel/traps.c` 中定义的。

第 431 行：为本地 APIC（如果使用的话）设置中断处理程序，并且为级联的 8259 调用 `irq.c` 中的 `setup_irq()`。

第 437 行：用 I/O 寄存器启动 8253 PIT。

第 419 行^①：在 `init/main.c` 代码中的第 419 行，遵照 `time_init()` 为 PPC 和 x86 安装系统时钟中断处理程序的函数。在 PPC 中，系统定时器初始化衰减器（为简单起见进行了缩简）：

```
-----
arch/ppc/kernel/time.c
void __init time_init(void)
{
...
317   ppc_md.calibrate_decr();
...
351   set_dec(tb_ticks_per_jiffy);
...
}
```

第 317 行：为系统的 HZ 值给出合适的数值。

第 351 行：为衰减器设置合适的初始值。

PowerPC 的中断体系结构及其在 Linux 中的实现不要求安装时钟中断。衰减器中断向量位于 0x900 处。处理程序的调用被硬编码在此处，并且不被共享。

```
-----
arch/ppc/kernel/head.S
/* Decrementer */
479  EXCEPTION(0x900, Decrementer, timer_interrupt, EXC_XFER_LITE)
-----
```

本节稍后将详细介绍操作衰减器的 `EXCEPTION` 宏。当达到终止计数时，衰减器的处理程序就做好被执行的准备。

下列代码片断简要描述 x86 系统定时器的初始化。

```
-----
arch/i386/kernel/time.c
void __init time_init(void)
{
...
340   time_init_hook();
}
```

函数 `time_init()` 调用 `time_init_hook()`，后者位于和机器相关的设置文件 `setup.c` 中。

```
-----
arch/i386/machine-default/setup.c
072 static struct irqaction irq0 = { timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};
...
081 void __init time_init_hook(void)
082 {
083   setup_irq(0, &irq0);
084 }
```

① 这一行代码和上面所列的代码也不是同一个文件，在 `init/main.c` 代码中调用，在 `arch/i386/kernel/time.c` 中实现。

第 72 行：初始化与 irq0 对应的 irqaction 结构体。

第 81~84 行：该函数调用 setup_irq(0, &irq0)，将包含处理程序 timer_interrupt() 的 irqaction 结构体放到与 irq0 相关的共享中断队列上。

对于一般的处理程序而言（那些不在内核初始化时装载的处理程序），这段代码与调用 request_irq() 的效果类似。时钟中断的初始化代码通过一个捷径把处理程序放入 irq_desc[] 中。运行时代码使用 irq.c 中的 disable_irq()、enable_irq()、request_irq() 和 free_irq()。所有这些函数都是能对 IRQ 起作用的实用程序，并在某一处涉及 irq_desc 结构。

● 中断时机

对于 PowerPC 而言，衰减器在处理器的内部，并让其中断向量位于地址 0x900 处。这和 x86 体系结构恰恰相反，在 x86 体系结构中，PIT 是一个来自于中断控制器的外部中断。PowerPC 的外部中断控制器位于向量 0x500 处。如果系统定时器是基于本地 APIC 的，那么 x86 体系结构中的情形也就和 PowerPC 中的情形类似了。

表 3-10 和表 3-11 分别描述 x86 和 PPC 体系结构的中断向量表。

表3-10 x86的中断向量表

中断向量号/IRQ	说 明
0	除法错误
1	调试扩展
2	NMI（不可屏蔽的）中断
3	断点
4	INTO指令检测到的溢出
5	BOUND检测到的越界
6	无效的操作码
7	设备不可用
8	检测到两个错误
9	协处理器段越界（保留）
10	无效的任务状态段
11	段不存在
12	栈错误
13	通用保护
14	缺页
15	（Intel预留，不要使用）
16	浮点错误
17	对齐检查
18	机器校验
19~31	（Intel预留，不要使用）
32~255	可屏蔽中断

表3-11 PPC中的中断向量偏移

偏移量 (十六进制)	中断类型
00000	保留
00100	系统复位
00200	机器校验
00300	数据存储
00400	指令存储
00500	外部中断
00600	对齐
00700	程序
00800	浮点不可用
00900	衰减器
00A00	保留
00B00	保留
00C00	系统调用
00D00	跟踪
00E00	浮点辅助
00E10	保留
...	...
00FFF	保留
01000	保留, 取决于具体实现
...	...
02FFF	(中断向量区域结束)

注意两种体系结构间的相似之处。这两个表描述的都是硬件。Intel 异常中断向量表的软件接口就是本章前面所提到的中断描述符表 (IDT)。

继续深入分析, 就会看到 Intel 体系结构处理硬件中断的全过程, 首先将 IRQ 传给处理器, 然后转到 entry.S 中的跳转表, 找到相应的调用门 (描述符), 最后获得处理程序的代码。图 3-19 说明了这个过程。

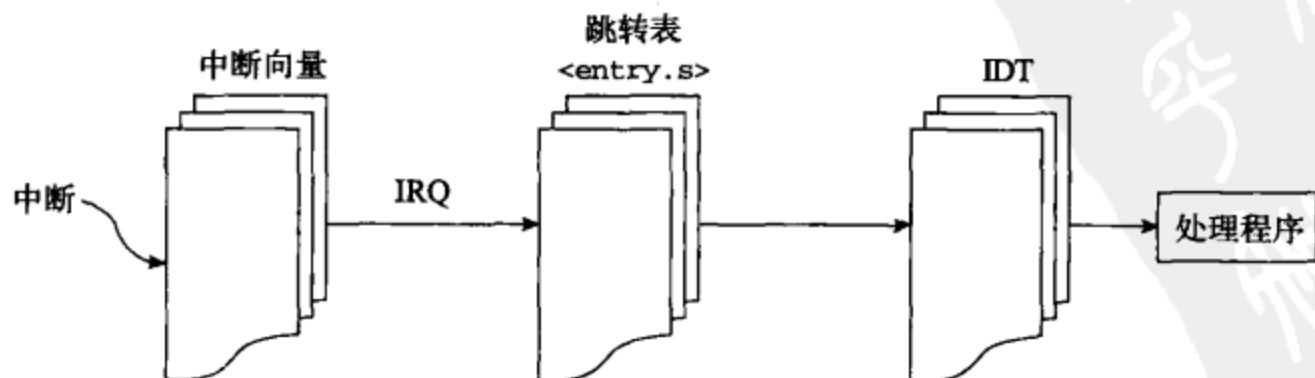


图 3-19 x86 的中断处理过程

另一方面, PowerPC 指向内存中明确的偏移处, 此处放置的是跳转到合适的中断处理程序的

代码。从图 3-20 中可以看到，位于 head.S 中的 PPC 跳转表通过在内存中被固定的方式被索引。图 3-20 展示了这一过程。

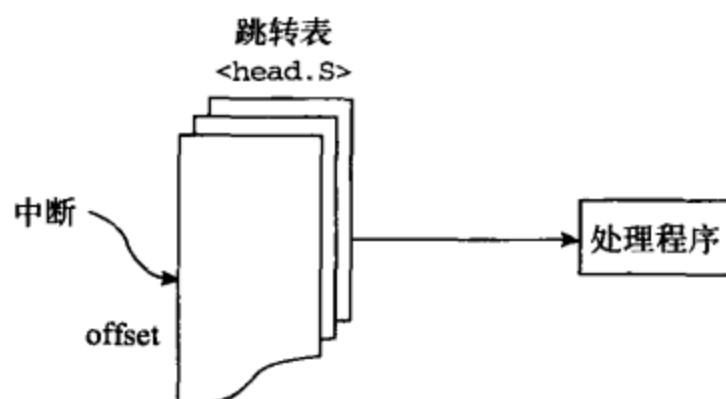


图 3-20 PPC 的中断处理过程

分析 PPC 的外部中断（偏移量 0x500）处理程序和定时器中断（偏移量 0x900）处理程序时，PPC 的中断处理过程会更清楚一些。

● 处理 PowerPC 外部中断向量

与前面所讨论的一样，如果外部中断发生，处理器就跳转到地址 0x500 处。进一步深入分析文件 head.S 中的 EXCEPTION() 宏后，可以看到下面几行代码被链接并装载，以便它被映射到偏移地址为 0x500 的这片内存区域。这种结构化的跳转表和 x86 的 IDT 作用相同：

```
-----
arch/ppc/kernel/head.S
453  /* External interrupt */
454  EXCEPTION(0x500, HardwareInterrupt, do_IRQ, EXC_XFER_LITE)

The third parameter, do_IRQ(), is called next. Let's take a look at this function.
arch/ppc/kernel/irq.c
510  void do_IRQ(struct pt_regs *regs)
511  {
512  int irq, first = 1;
513  irq_enter();
...
523  while ((irq = ppc_md.get_irq(regs)) >= 0) {
524  ppc_irq_dispatch_handler(regs, irq);
525  first = 0;
526  }
527  if (irq != -2 && first)
528  /* That's not SMP safe ... but who cares ? */
529  ppc_spurious_interrupts++;
530  irq_exit();
531  }
-----
```

第 513~530 行：表示正处于硬件中断中的抢占代码。

第 523 行：从中断控制器读取一个待处理的中断，并转换成 IRQ 号（直至所有的中断都被处理）。

第 524 行：ppc_irq_dispatch_handler() 处理中断。下面将更详细地分析这个函数。

函数 `ppc_irq_dispatch_handler()` 几乎等同于 x86 的函数 `do_IRQ()`。

```
-----
arch/ppc/kernel/irq.c
428 void ppc_irq_dispatch_handler(struct pt_regs *regs, int irq)
429 {
430     int status;
431     struct irqaction *action;
432     irq_desc_t *desc = irq_desc + irq;
433
434     kstat_this_cpu.irqs[irq]++;
435     spin_lock(&desc->lock);
436     ack_irq(irq);
437
438     ...
441     status = desc->status & ~(IRQ_REPLAY | IRQ_WAITING);
442     if (!(status & IRQ_PER_CPU))
443         status |= IRQ_PENDING; /* we _want_ to handle it */
444
445     ...
449     action = NULL;
450     if (likely(!(status & (IRQ_DISABLED | IRQ_INPROGRESS)))) {
451         action = desc->action;
452         if (!action || !action->handler) {
453             ppc_spurious_interrupts++;
454             printk(KERN_DEBUG "Unhandled interrupt %x, disabled\n", irq);
455             /* We can't call disable_irq here, it would deadlock */
456             ++desc->depth;
457             desc->status |= IRQ_DISABLED;
458             mask_irq(irq);
459             /* This is a real interrupt, we have to eoi it,
460              so we jump to out */
461             goto out;
462         }
463         status &= ~IRQ_PENDING; /* we commit to handling */
464         if (!(status & IRQ_PER_CPU))
465             status |= IRQ_INPROGRESS; /* we are handling it */
466     }
467     desc->status = status;
468
469     ...
489     for (;;) {
490         spin_unlock(&desc->lock);
491         handle_irq_event(irq, regs, action);
492         spin_lock(&desc->lock);
493
494         if (likely(!(desc->status & IRQ_PENDING)))
495             break;
496         desc->status &= ~IRQ_PENDING;
497     }
498 out:
499     desc->status &= ~IRQ_INPROGRESS;
500
501     ...
511 }
```

第 432 行：从参数获得 IRQ 并获取对适当 `irq_desc` 的访问。

第 435 行：获得 IRQ 描述符上的自旋锁，以防同一个中断被不同的 CPU 并发访问。

第 436 行：给硬件发送一个确认，随后，硬件给出回应，在这个中断处理完之前，防止更多该类型的中断被处理。

第 441~443 行：标志 IRQ_REPLAY 和 IRQ_WAITING 被清零。在这种情况下，IRQ_REPLAY 表示该 IRQ 曾经被丢弃，现在准备重新发送，IRQ_WAITING 表示 IRQ 正在被测试。（这两种情况都超出了本书的讨论范围。）在单处理器系统中，设置 IRQ_PENDING 标志，表示答应负责处理中断。

第 450 行：这段代码检查不处理中断的情形。如果设置了 IRQ_DISABLED 或 IRQ_INPROGRESS 就可以忽略这段代码。当不想让系统响应正在处理的特定 IRQ 线时，就设置 IRQ_DISABLED。IRQ_INPROGRESS 表示中断正在被一个处理器处理，这用在多处理器系统的第二个处理器试图产生同一个中断的情况。

第 451~462 行：此处查看中断处理程序是否存在。如果不存在，就跳转到第 498 行的标号“out”处。

第 463~465 行：此处将三个不处理中断的条件全部清除，因此允许处理中断，即设置 IRQ_INPROGRESS 标志，清除 IRQ_PENDING 标志，表示中断正在处理。

第 489~497 行：中断得到处理，在处理中断之前，释放中断描述符上的自旋锁。自旋锁被释放后，将调用 handle_irq_event() 例程。这个例程将执行中断处理程序。执行完毕后就会再次获得描述符上的自旋锁。如果在 IRQ 处理期间没有其他 CPU 设置 IRQ_PENDING 标志，就跳出循环。否则，再次处理中断。

● 处理 PowerPC 的系统时钟中断

在 timer_init() 中已经指出，衰减器被硬编码到地址 0x900 处。假定已经达到终止计数，此时将调用 arch/ppc/kernel/time.c 中的处理程序 timer_interrupt()。

```
-----
arch/ppc/kernel/head.S
/* Decrementer */
479 EXCEPTION(0x900, Decrementer, timer_interrupt, EXC_XFER_LITE)
-----
```

这是 timer_interrupt() 函数。

```
-----
arch/ppc/kernel/time.c
145 void timer_interrupt(struct pt_regs * regs)
146 {
...
152 if (atomic_read(&ppc_n_lost_interrupts) != 0)
153     do_IRQ(regs);
154
155 irq_enter();
...
159 if (!user_mode(regs))
160     ppc_do_profile(instruction_pointer(regs));
...
165 write_seqlock(&xtime_lock);
```

```

166
167     do_timer(regs);
...
189     if (ppc_md.set_rtc_time(xtime.tv_sec+1 + time_offset) == 0)
...
195     write_sequnlock(&xtime_lock);
...
198     set_dec(next_dec);
...
208     irq_exit();
209 }

```

第 152 行：如果中断丢失，则回去调用 0x900 处的外部中断处理程序。

第 159 行：对内核路径调试进行内核分析。

第 165 和 195 行：把这段代码锁在外面。

第 167 行：这行代码和 x86 时钟中断使用的是同一个函数（下面将讨论）。

第 189 行：更新 RTC。

第 198 行：为下一个中断重新启动衰减器。

第 208 行：从中断返回。被中断的代码现在正常运行，直到下一个中断出现。

● 处理 x86 的系统时钟中断

中断激活时（在本例中，PIT 倒计时到 0 并激活 IRQ0），中断控制器激活连入处理器的中断请求线。entry.S 中的汇编代码有对应于 IDT 中每个描述符的入口点。IRQ0 是第一个外部中断，在 IDT 中是向量 32。于是，这段代码准备跳转到位于 entry.S 中的跳转表的入口点 32 处。

```

-----
arch/i386/kernel/entry.S
385     vector=0
386     ENTRY(irq_entries_start)
387     .rept NR_IRQS
388     ALIGN
389     1:     pushl $vector-256
390     jmp common_interrupt
391     .data
392     .long 1b
393     .text
394     vector=vector+1
395     .endr
396
397     ALIGN
398     common_interrupt:
399     SAVE_ALL
400     call do_IRQ
401     jmp ret_from_intr
-----

```

这是一段精彩的汇编程序。重复结构 .rept（在第 387 行）以及结束语句（在第 395 行）在编译时创建中断跳转表。注意，当这个代码块重复创建时，第 389 行压入栈的向量号是递减的。把向量压入栈，内核代码就知道，中断发生时它正在对哪一个 IRQ 进行处理。

当离开对 x86 代码的跟踪时，代码就跳转到跳转表中适当的入口点，并把 IRQ 存放到栈中，然后跳转到第 398 行的公用处理程序，并调用第 400 行的 `do_IRQ()` (`arch/i386/kernel/irq.c`)。这个函数几乎和 `ppc_irq_dispatch_handler()` 完全相同，`ppc_irq_dispatch_handler()` 在本节的“处理 PowerPC 外部中断向量”中已经描述过，此处不再赘述。

函数 `do_irq()` 可以根据传入的 IRQ 来访问 `irq_desc` 中的适当元素，并跳转到 `action` 结构链表中的每个中断处理程序。此处，实际上是让它成为 PIT 的处理函数：`timer_interrupt()`，参见 `time.c` 中的下列代码段。处理程序从第 274 行处开始，并保持和源文件中相同的次序：

```
-----
arch/i386/kernel/time.c
274 irqreturn_t timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
275 {
...
287 do_timer_interrupt(irq, NULL, regs);
...
290 return IRQ_HANDLED;
291 }
-----
```

第 274 行：这是系统时钟中断处理程序的入口点。

第 287 行：这是调用函数 `do_timer_interrupt()`。

```
-----
arch/i386/kernel/time.c
208 static inline void do_timer_interrupt(int irq, void *dev_id,
209 struct pt_regs *regs)
210 {
...
227 do_timer_interrupt_hook(regs);
...
250 }
-----
```

第 227 行：调用函数 `do_timer_interrupt_hook()`。这个函数本质上是对 `do_timer()` 调用的封装。让我们来看看：

```
-----
include/asm-i386/mach-default/do_timer.h
016 static inline void do_timer_interrupt_hook(struct pt_regs *regs)
017 {
018 do_timer(regs);
...
025 x86_do_profile(regs);
...
030 }
-----
```

第 18 行：调用函数 `do_timer()`。这个函数完成更新系统时间的大部分工作。

第 25 行：`x86_do_profile()` 程序用于查看 `eip` 寄存器，`eip` 寄存器存放中断前正在运行的代码。这个数据表示，随着时间的推移，进程运行的频繁程度。

在这里，系统时钟中断从 `do_irq()` 返回到 `entry.S` 进行系统事务管理，被中断的线程恢复执行。

如前所述，系统定时器是 Linux 操作系统的脉搏。尽管本章中定时器作为中断的一个实例，实际上它的用途遍及整个操作系统。

3.9 小结

进程必须和其他进程共享处理器并定义各自的执行上下文，执行上下文包含进程运行所必须的所有信息。在它们的执行期间经历了各种状态，这些状态可被抽象成阻塞状态、运行状态和就绪状态。

内核把有关进程的信息存放在 `task_struct` 描述符中，根据进程所具有的不同功能，可以把 `task_struct` 的字段划分为：进程的属性、进程间的关系、进程的内存访问、与进程相关的文件操作、进程的信任状、资源限制以及调度。所有这些字段都是记录进程上下文所必须的。进程可由一个或多个共享内存地址空间的线程组成。每个线程都有自己的结构。

通过调用 `fork()`、`vfork()` 或 `clone()` 中的某一个系统调用，一个新的进程就此诞生。这三个系统调用最终都调用了内核例程 `do_fork()`，`do_fork()` 完成新进程创建的大部分工作。在执行期间，进程从一个状态转换到另一个状态。进程经由调度程序的选择从就绪状态转换到运行状态；如果它的时间片用完或者如果它把 CPU 让给其他进程，就会从运行状态转换到就绪状态；如果等待的信号到来，就从阻塞状态转换到就绪状态；当进程等待一个资源或进程睡眠时，就从运行状态转换到阻塞状态。进程的消亡发生在调用 `exit()` 时。

本章还深入研究调度程序的基本架构以及它所使用的数据结构，包括运行队列、等待队列以及怎样管理这些结构以便记录进程被调度的踪迹。

通过研究 x86 和 PPC 的硬件如何处理中断，本章最后讨论了进程的异步执行流程（包括异常和中断）。我们以系统定时器为例，分析了 Linux 内核如何管理硬件提交的中断。

3.9.1 项目：系统变量 `current`

本章分析了 `task_struct` 和 `current`。`current` 是系统变量，它指向当前执行进程的 `task_struct`。本项目的目标是加强这样一个概念，即内核是一系列有序而不断变化的连环结构，随着程序运行这些结构被创建和销毁。如我们所见，`task_struct` 结构是内核所拥有的最重要的结构之一，其中含有给定进程所必须的全部信息。这个项目模块如同内核那样访问 `task_struct` 结构，并为读者今后的进一步研究打下基础。

在本项目中，将访问当前进程的 `task_struct` 并输出该结构的各种元素。从文件 `include/linux/sched.h` 出发，直到找到 `task_struct`。可以借助文件 `sched.h` 来引用 `current->pid`、`current->comm` 和当前进程的 ID 及其名字，并沿着这些结构找到父进程的 `pid` 和 `comm`。接下来，借用 `printk()` 的一个例程为正在使用的 tty 终端回送一个消息。

请看下列代码。

注意 从运行第一个程序 (`hellomod`) 开始，当初始化例程打印出 `current->comm` 时，当前进程的名字会是什么呢？父进程的名字又会是什么呢？请看下列代码的分析。

3.9.2 项目源码^①

```

-----
currentptr.c
001  #include <linux/module.h>
002  #include <linux//kernel.h>
003  #include <linux/init.h>
004  #include <linux/sched.h>
005  #include <linux/tty.h>
006
007      void tty_write_message1(struct tty_struct *, char *);
008
009      static int my_init( void )
010      {
011
012          char *msg="Hello tty!";
013
014          printk("Hello, from the kernel...\n");
015          printk("parent pid =%d(%s)\n",current->parent->pid,current->
parent->comm);
016          printk("current pid =%d(%s)\n",current->pid,current->comm);
017
018          tty_write_message1(current->signal->tty,msg);
019          return 0;
020      }

022  static void my_cleanup( void )
{
    printk("Goodbye, from the kernel...\n");
}

027  module_init(my_init);
028  module_exit(my_cleanup);

// This routine was borrowed from <printk.c>
032  void tty_write_message1(struct tty_struct *tty, char *msg)
{
    if (tty && tty->driver->write)
        tty->driver->write(tty, 0, msg, strlen(msg));
    return;
037  }
-----

```

第 4 行: sched.h 包含 struct task_struct{}, 我们从这里引用进程的 ID (->pid) 和当前进程的名字(->comm), 以及指向父进程 PID (->parent) 的指针, 该指针指向父进程的 task 结构。同样也会找到指向信号结构的指针, 信号结构包含对 tty 结构的引用 (参见第 18~22 行)。

第 5 行: tty.h 包含 struct tty_struct{}, 我们从 printk.c 借用的例程要使用它 (见第 32~37 行)。

第 12 行: 这是将回送给终端的简单消息字符串。

第 15 行: 此处从当前进程的 task 结构引用其父进程的 PID 及其名字。前面问题的答案是:

^① 你可以把项目源码作为探究正在运行的内核的起点。尽管可以使用内核中许多有用的例程来查看信息, 如内部函数 (例如, strace()), 但构建如本项目这样的自定义工具, 可以展现出 Linux 内核实时性的一面。

该进程的父进程是当前的 shell 程序，在本例中，它就是 Bash。

第 16 行：此处从当前进程的 task 结构引用它的 PID 及其名字。为了回答前面问题的前半部分，只要在 Bash 命令行上输入 `insmod`，它就会被作为当前进程打印出来。

第 18 行：这是从 `kernel/printk.c` 借用的函数。它用于把消息重定向到指定的 tty。为了说明 `current` 的要点，可以从调用我们程序的地方把 tty（窗口或命令行）的 `tty_struct` 传递给这个例程，这是通过 `current->signal->tty` 来引用的。参数 `msg` 是在第 12 行声明的字符串。

第 32~38 行：`tty write` 函数检查到 tty 的存在，接着，以消息为参数调用适当的设备驱动程序。

3.9.3 运行代码

编译，并像第一个项目那样 `insmod()` 代码。

3.10 习题

1. 描述进程状态时，我们把“等待或阻塞”状态描述为进程没有运行也没有就绪时所处的状态。等待和阻塞之间有什么区别？在什么条件下进程处于等待状态，什么条件下它处于阻塞状态？

2. 找出把进程从运行状态设置为阻塞状态的内核代码。换句话说，找出 `current->state` 的状态从 `TASK_RUNNING` 转换到 `TASK_STOPPED` 的地方。

3. 为了弄清楚计数器的“倒计时”花费多少时间，请完成下面的计算。如果 64 位的衰减器以 500MHz 运行，终止下列值要用多长时间？

- a. `0x000000000000ffff`
- b. `0x00000000ffffffff`
- c. `0xffffffffffffff`

4. 当某段代码不应该被中断时，老版本的 Linux 是利用 `sti()` 和 `cli()` 来禁用中断的，而新版本的 Linux 使用 `spin_lock()`。自旋锁的主要优势是什么？

5. x86 的例程 `do_IRQ()` 和 PPC 的例程 `ppc_irq_dispatch_handler()` 如何处理共享的中断？

6. 为什么不推荐由内核代码访问系统调用？

7. 在运行 2.6 内核的 Linux 系统中，每个 CPU 有几个运行队列？

8. 当进程创建新进程时，Linux 要求它放弃一部分时间片吗？如果是，为什么？

9. 在进程的时间片到期后，怎样把它重新插入运行队列的优先级数组中？进程优先级的正常范围是多少？实时进程的优先级范围呢？

第4章

内存管理

本章内容

- ☐ 页
- ☐ 内存管理区
- ☐ 页面
- ☐ Slab 分配器
- ☐ Slab 分配器的生命周期
- ☐ 内存请求路径
- ☐ Linux 进程的内存结构
- ☐ 进程映像的分布及线性地址空间
- ☐ 页表
- ☐ 缺页

内存管理是运行在计算机上的应用程序通过软硬件协作来访问内存的一种方法。内存管理子系统的职责为：进程请求内存时分配可用内存，进程释放内存后回收内存，以及跟踪系统中内存的使用状况。

操作系统的生命周期划分为两个阶段：正常执行阶段和自举阶段。自举阶段使用临时内存；而正常运行阶段使用的内存有两种情况：一种是有一部分固定的内存分配给内核代码和数据，另一种是为动态内存请求分配内存。动态内存请求源于进程的创建和空间的扩张。本章要着重介绍操作系统正常运行时对内存的管理。

在深入研究内存管理的实现细节和各部分如何配合前，很有必要先理解一些有关内存管理的高级概念。本章首先介绍内存管理系统和虚拟内存的概念，然后介绍辅助内存管理的各种内核数据结构和算法。理解了内核如何管理内存之后，接着还要考虑进程内存是如何划分与管理的，并概括出它们是如何以自顶向下的方式组织到内核数据结构中的。最后当介绍完进程内存的获取、管理和释放后，我们会讨论缺页，以及上述这些概念在 PowerPC 和 x86 两种体系结构上的具体实现。

最简单的内存管理系统是运行进程对所有内存具有访问权的系统。以这样方式运行的进程必须包含对所需要的硬件进行操作的全部代码，必须能找到自己的内存地址，而且还必须能将自身的数据载入内存。这种方式不但给程序开发人员造成了很重的负担，而且还要保证进程与可用内存的大小合适。这些苛刻的要求对于日益复杂化的程序需求来说显然很不现实，所以要将内存管

理这个棘手的任务交给操作系统来对付，可用内存会在操作系统和用户进程之间划分。

当代操作系统既要求能够使多个程序共享系统资源，同时还要求内存限制对程序开发者透明。在此需求下，**虚拟内存**（Virtual memory）应运而生，虚拟内存支持程序访问比系统物理可用内存大得多的内存空间，而且也使得多个程序共享内存变得更高效。物理内存（或叫核心内存）是系统中由 RAM 芯片控制的可用内存。虚拟内存依靠透明地使用磁盘空间，得以使程序运行起来好像它们使用比系统物理内存更多的内存空间。磁盘空间（相比物理内存价格更低廉，容量也更大）可作为物理内存的扩充。我们之所以称其为虚拟内存就是因为磁盘存储体有效地充当内存，但它本身并不是内存。图 4-1 描述了多级数据存储体的层次关系。

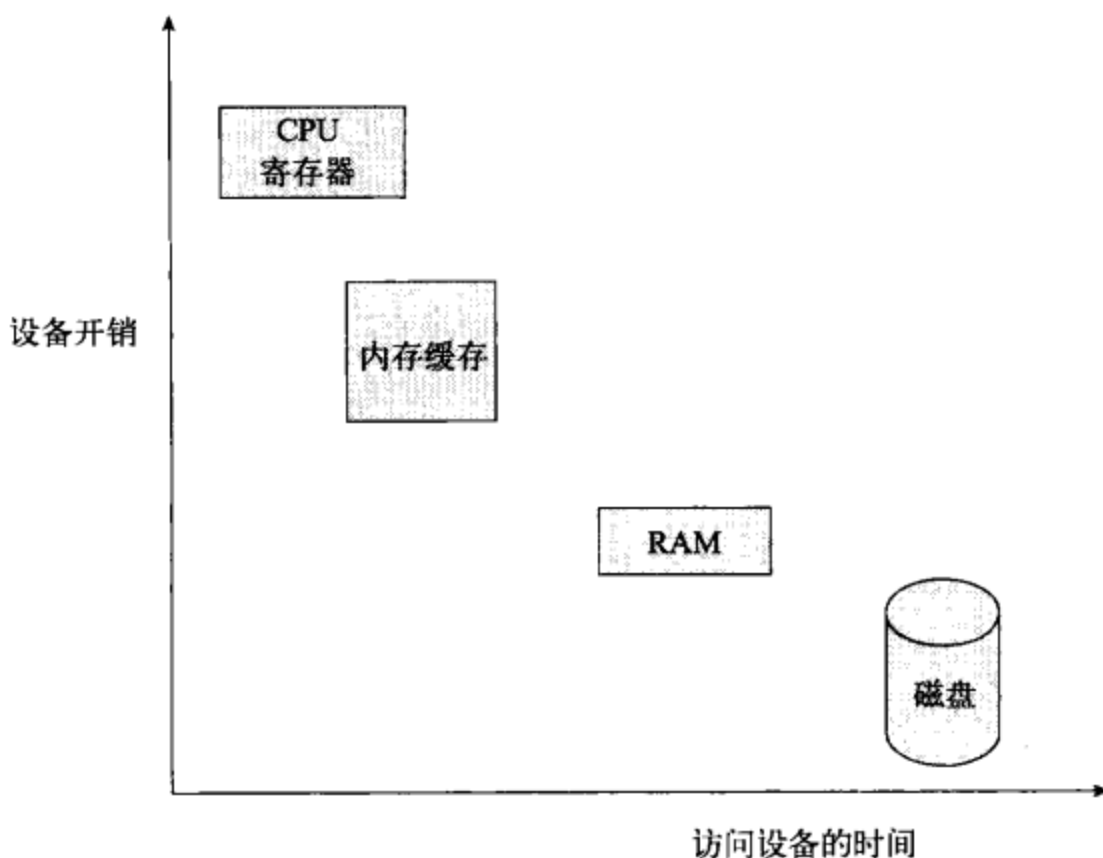


图 4-1 数据访问层次图

使用虚拟内存时，程序数据被分割成基本单元，这些单元可以在磁盘与内存间来回移动。这样，程序正使用的那部分就可以置于内存中，以便快速被访问，而未用的部分则被临时存放在磁盘，如此一来减轻待访问数据存在磁盘上导致读取时间过长的问题。这些数据单元（或者说虚拟内存块）被称作**页**（Page）。同理，物理内存也需要被划分成用于保存这些页的区，这些区被称作**页面**（Page Frame）。当进程请求访问一个地址时，该地址所在的页被载入内存，对页中任一数据的请求都会产生对该页的访问。如果页中的任一地址以前都没有被访问过，说明该页尚未被装入内存。对页中地址第一次访问便会产生一个**失败或缺页**（page fault），因为这时该页并没有在内存中，因此必须从磁盘请求。一次缺页便是一个**陷入**（trap）。当发生缺页时，内核必须选择一个页面，然后将其内容（页）写回到磁盘，从而用程序刚刚请求的页的内容来填充它。

当程序从内存中存取数据时，会使用地址来指出需要访问的内存位置。该地址被称作**虚拟地址**（virtual address），它们组成了进程**虚拟地址空间**（virtual address space）。每个进程都有自己独立的虚拟地址范围，这样做的好处是可防止非法读取或写覆盖其他进程的数据。虚拟内存允许

进程“使用”超过可用物理内存的内存空间，因此操作系统可以给每个进程提供独立的虚拟线性地址空间^①。

虚拟地址空间的大小取决于体系结构的字长，如果处理器的寄存器可容纳 32 位数值，那么运行该程序的处理器支持的进程虚拟地址空间由 2^{32} 个地址构成^②。虚拟内存不但扩大了可寻址的内存范围，而且也使得用户空间的开发者不必担心物理内存的本质所带来的限制，比如开发者不需要管理内存中的任何漏洞。以 32 位的系统为例，其虚拟地址空间的范围是 0~4 G，如果系统有 2 G 的物理内存，那么它的物理地址范围是 0~2 G。而程序可能有 4 GB 之巨，但是必须被装入可用内存中才能运行，因此整个程序将被存放在磁盘上，只有当需要时页才会被载入到内存。

将页在内存到磁盘之间调入调出的机制被称作**分页机制** (paging)，分页包括程序虚拟地址到物理内存地址的转换。

内存管理器在操作系统中负责维护虚拟地址和物理地址之间的关系，并且实现分页机制。对内存管理来说，页是基本的内存单元；MMU (Memory Management Unit，**内存管理单元**) 是完成实际的地址转换工作^③的硬件部件；内核提供了**页表** (对可用页进行索引的列表) 以及 MMU 在执行地址转换时要访问的相关地址。上述这些数据都会在页面载入内存时被更新。

我们了解内存管理的这些高层概念后，接下来开始分析内核如何实现内存管理器，先来看看页的实现。

4.1 页

作为内存管理器管理的基本内存单元，页的许多状态需要被记录下来。比如内核需要知道什么时候页可以被回收。为此，内核使用**页描述符**，内存中每个物理页都对应一个页描述符。

这一节阐述页描述符的各种字段，并对内存管理器如何使用这些字段给出说明。页结构定义在 include/linux/mm.h 文件中。

```
-----
include/linux/mm.h
170 struct page {
171     unsigned long flags;
172
173     atomic_t count;
174     struct list_head list;
175     struct address_space *mapping;
176     unsigned long index;
177     struct list_head lru;
178
179     union {
180         struct pte_chain *chain;
```

① 关于进程内存的使用，进程寻址做出几条假设：第一，进程不必同时使用其请求的所有内存；第二，从一个普通的可执行文件实例化而产生的两个或多个进程只需要将可执行对象载入内存一次。

② 虽然从技术上讲可用内存的大小界限是物理内存加上 swap 空间，但可寻址范围则取决于体系结构的字长。这意味着即便系统中有超过 4 GB 的内存，进程分配的内存也不超过 3 GB (要知道顶端 1 GB 是分配给内核的空间)。

③ 有些微处理器，比如 Motorola 68000 (68 K)，缺少 MMU 单元。uClinux 是一种特殊的 Linux 发布版，它被专门移植到非 MMU 系统中运行。它没有 MMU，因此虚拟地址等同于物理地址。

```

181
182     pte_addr_t direct;
183 } pte;
184 unsigned long private;
185
...
196 #if defined(WANT_PAGE_VIRTUAL)
197     void *virtual;
198
199 #endif
200 };
-----

```

标志

原子标志描述页面的状态。每个标志都由 32 位数值中的一位表示。系统提供一组辅助函数，用于操作和测试特定的标志，同时还可以通过一组辅助函数访问对应于特定标志的位的值。这些标志和辅助函数都定义于 `include/linux/page-flags.h` 文件中，表 4-1 给出并解释了可在页结构的 `flags` 字段中设置的一些标志。

表4-1 page->flags中的标志值

标 志 名	描 述
PG_locked	表示页被锁定，因此不可访问。该位用于保护磁盘I/O操作，在I/O操作前设置，I/O操作完成后清除
PG_error	表示这一页有I/O错误发生
PG_referenced	表示这一页在磁盘I/O操作期间被访问过。该标记可以决定页应位于哪个活动页链表上或非活动页链表上
PG_uptodate	表示页内容是有效的，当在页上的读操作完成后，该标志被设置。该标志与PG_error标志互斥
PG_dirty	表示被修改的页
PG_lru	表示该页是某个用于页交换的LRU（最近最少使用）链表。请看本节中对页结构lru字段的描述，从而了解更多关于LRU链表的信息
PG_active	表示页处于活动页链表中
PG_slab	表示页属于由slab分配器创建的Slab。其详细描述请见4.4节
PG_highmem	表示页位于高端内存区（ZONE_HIGHMEM）中。它不可被永久映射到内核虚拟地址空间中，高端内存区中的页在内核启动时由mem_init()函数打上该标志（请看第8章了解具体细节）
PG_checked	由ext2文件系统使用，但在2.5版内核中摒弃了该标志
PG_arch_1	与体系结构相关的页状态位
PG_reserved	表示该页不可被换出、不存在，或者已被启动内存分配器分配
PG_private	表示该页是有效的。当page->private包含有效值时，设置该标志
PG_writeback	表示页正在被写回
PG_mappedtodisk	表示页中含有当前在系统磁盘上分配的块区
PG_reclaim	表示页应该被回收
PG_compound	表示该页是高级混合页的一部分

1. count

count 字段相当于计数器，统计一个页使用或引用的次数。数值 0 表示页面是可重用的；正数表示访问该页数据^①的进程数。

2. list

list 字段是 list_head 结构，结构中的 next 和 prev 指针指向双向链表中的对应元素。这个页是否是双向链表的成员，其部分因素是由与页相关的映射和页状态确定的。

3. mapping

当页中存放的数据对应着文件的内存映射时，其中每个页就关联一个 address_space 结构。mapping 字段是指向 address_space 结构的指针，而页是属于该结构的成员。address_space 其实是属于内存对象（比如索引节点）的页面集合。有关如何使用 address_space 的详细信息，请参见第 7 章。

4. lru

lru 字段存放的 next 和 prev 指针指向最近最少使用链表中的相应元素。这些链表用在页面回收处理中，包括两个相应的链表：活动链表（active_list）和非活动链表（inactive_list），活动链表包含在用的页面，非活动链表包含可重用的页面。

5. virtual

virtual 是一个指向页所对应虚拟地址的指针。在含有高端内存^②的系统中，内存映射可以动态完成，所以很有必要在需要时重新计算虚拟地址，在这种情况下，该值要被设置成 NULL。

混合页

混合页是一种高阶页。如果能让内核支持混合页，就需要在编译时启用“Huge TLB Page Support”项。一个混合页由多个页面组成，其中首页被称为“头”页，其余被称为“尾”页。所有的混合页都需要在其相应的 page->flags 中设置 PG_compound 位，而且 page->lru.next 指向头页。

4.2 内存管理区

并非所有页都会被平等对待。在有些计算机体系中，对内存中某一物理地址范围的用途是有限制的。比如在 x86 体系中，部分 ISA 总线只能寻址最开始 16 MB 的内存范围。虽在 PPC 没有此类限制，但内存管理区（memory zone）的概念还是被保留下来，用以简化与体系结构无关的代码。因此在 PPC 的与体系结构相关的代码中，这些区被设置为重叠的。当一个系统的物理内存大于它能寻址的线性地址空间时，还会存在另一些此类限制。

内存管理区是由页面（或叫物理页）组成，这意味着，页面的分配来自特定的内存管理区。在 Linux 系统中存在三个内存管理区：ZONE_DMA（用于 DMA 的页面），ZONE_NORMAL（具有虚拟映射的非 DMA 页），ZONE_HIGHMEM（这些页的地址不在虚拟地址空间中）。

① 当页中的数据不再使用或不再需要时，该页便是自由的。

② 高端内存指超过了虚拟地址范围的物理内存部分。请参见 4.2 节了解更多细节。

4.2.1 内存管理区描述符

与内核管理的所有对象一样，每个内存管理区都对应一个叫做区域（zone）的结构，其中存放有关内存管理区的所有信息。区域结构定义于/linux/mmzone.h 文件中。接下来进一步学习一下该结构中最常用的一些字段。

```
-----
include/linux/mmzone.h
66 struct zone {
...
70  spinlock_t    lock;
71  unsigned long  free_pages;
72  unsigned long  pages_min, pages_low, pages_high;
73
74  ZONE_PADDING(_pad1_)
75
76  spinlock_t    lru_lock;
77  struct list_head active_list;
78  struct list_head inactive_list;
79  atomic_t      refill_counter;
80  unsigned long  nr_active;
81  unsigned long  nr_inactive;
82  int            all_unreclaimable; /* All pages pinned */
83  unsigned long  pages_scanned; /* since last reclaim */
84
85  ZONE_PADDING(_pad2_)
...
103  int temp_priority;
104  int prev_priority;
...
109  struct free_area free_area[MAX_ORDER];
...
135  wait_queue_head_t * wait_table;
136  unsigned long      wait_table_size;
137  unsigned long      wait_table_bits;
138
139  ZONE_PADDING(_pad3_)
140
...
157 } ____cacheline_maxaligned_in_smp;
-----
```

1. lock

当读写内存管理区描述符时，必须对其锁定以避免读/写错误。lock 字段存放的自旋锁便是保护描述符的。

该锁保护的是内存管理区描述符本身，而不是描述符所描述的内存范围。

2. free_pages

free_pages 字段存放内存管理区中剩余的空闲页个数。每当从某个内存管理区中分配一个页面，或者向内存管理区返回一个页面时，就对这个无符号长整数分别执行加 1 或者减 1 操作。

调用 `nr_free_pages()` 函数所返回的系统空闲总页数，就是系统中三个内存管理区中的该值相加的总和。

3. `pages_min`、`pages_low`和`pages_high`

`pages_min`、`pages_low` 和 `pages_high` 字段存放内存管理区的水位值（阈值）。当可用页面数跌到上述值时，意味着出现了内存紧张的紧急情况，此刻内核会根据相应规则做出响应。

4. `lru_lock`

`lru_lock` 字段存放的是空闲页列表的自旋锁。

5. `active_list`和`inactive_list`

`active_list` 和 `inactive_list` 两个链表用在页面回收处理时，前一个是处于活动状态的页链表，后一个是可被回收的页链表。

6. `all_unreclaimable`

如果内存管理区中所有页都被固定住，那么 `all_unreclaimable` 字段被置为 1。这些页只能被 `kswapd`（页换出守护进程）回收。

7. `pages_scanned`、`temp_priority`和`prev_priority`

`pages_scanned`、`temp_priority` 和 `prev_priority` 字段用于页面回收处理。它们的用法已经超出本书范围。

8. `free_area`

使用 `free_area` 位图的伙伴系统。

9. `wait_table`、`wait_table_size`和`wait_table_bits`

`wait_table`、`wait_table_size` 和 `wait_table_bits` 字段与内存管理区上页的进程等待队列相关。

缓存对齐和区填充

缓冲对齐是为了提高对描述符中字段的访问性能。缓冲对齐通过减少复制一组数据所需的指令数量来提高性能。假设有一个 32 位数值未按照字长对齐，那么处理器就需执行两次“载入字”的指令才把数据置入寄存器中，如果对齐的话，仅需要一次。`ZONE_PADDING` 字段描述内存管理区中缓冲的对齐方式。

```
-----
include/linux/mmzone.h
#ifdef CONFIG_SMP
struct zone_padding {
    int x;
} ____cacheline_maxaligned_in_smp;
#define ZONE_PADDING(name) struct zone_padding name;
#else
#define ZONE_PADDING(name)
#endif
-----
```

如果你想知道 Linux 中缓存对齐的工作原理，请参考 `include/linux/cache.h` 文件。

4.2.2 内存管理区操作辅助函数

当对一个对象进行一些常用的操作时，或者从对象中获取信息时，辅助函数通常可以简化编码。下面我们列出一些简化内存管理区操作的辅助函数。

1. `for_each_zone()`

宏 `for_each_zone()` 遍历系统中的所有内存管理区。

```
-----
include/linux/mmzone.h
268 #define for_each_zone(zone) \
269     for (zone = pgdat_list->node_zones; zone; zone = next_zone(zone))
-----
```

2. `is_highmem()`和`is_normal()`

`is_highmem()`和 `is_normal()` 两个函数检测内存管理区结构位于高端内存管理区中还是普通内存管理区中，其原型分别如下：

```
-----
include/linux/mmzone.h
315 static inline int is_highmem(struct zone *zone)
316 {
317     return (zone - zone->zone_pgdat->node_zones == ZONE_HIGHMEM);
318 }
319
320 static inline int is_normal(struct zone *zone)
321 {
322     return (zone - zone->zone_pgdat->node_zones == ZONE_NORMAL);
323 }
-----
```

4.3 页面

页面 (page frame) 是存放页的基本内存单元。只要进程请求内存，内核便会请求一个页面给它；同样道理，如果页面不再被使用，那么内核便将其释放，以便其他进程可以再使用。下面介绍的函数用来完成这些操作。

4.3.1 请求页面的函数

可以调用几个例程请求页面。我们依据其返回值的类型可以将它们分为两类。第一类返回指向 `page` 结构体的指针 (返回类型是 `void *`)，该结构体对应分配给请求者的页面。这类函数包括 `alloc_pages()` 和 `alloc_page()`。第二类函数返回 32 位虚拟地址 (返回类型是长整型)，该地址是所分配页面的首地址。这类函数包括 `__get_free_page()` 和 `__get_dma_pages()`。上述这些例程很多是对底层接口的包装。图 4-2 和图 4-3 描述这些例程的调用图。

接下来的宏和函数所请求或释放的页面数都是 2 的幂次方。也就是说，页面请求和释放都必须针对连续的页面，而且这些连续页面数目必须是 2 的幂次方。所以我们可以请求 1 页、2 页、4 页、8 页、16 页 (依次类推) 等数量页面^①。

① 页面的请求或释放总是连续的。

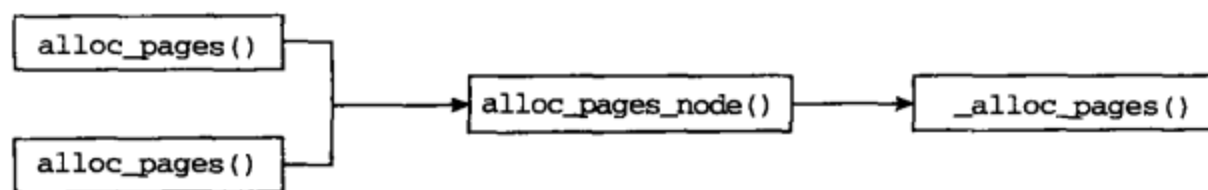


图 4-2 alloc_*()调用图

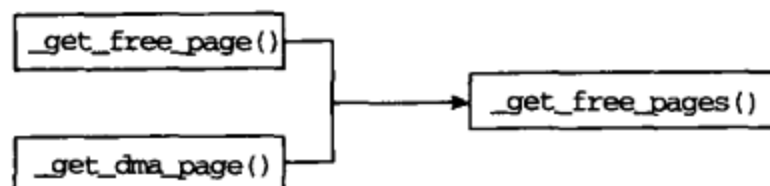


图 4-3 get_*_page()调用层次结构

1. alloc_pages()和alloc_page()

函数 alloc_page()用于请求单页，所以不需要说明内存的大小（不使用 order 参数）。该函数在调用 alloc_pages_node()时为 order 参数填充 0 值。而另一个函数 alloc_pages()则可以请求 4 个页面。

```

-----
include/linux/gfp.h
75  #define alloc_pages(gfp_mask, order) \
76    alloc_pages_node(numa_node_id(), gfp_mask, order)
77  #define alloc_page(gfp_mask) \
78    alloc_pages_node(numa_node_id(), gfp_mask, 0)
-----
  
```

正如我们在图 4-2 中所看到的，上述两个宏其实都是调用函数__alloc_pages_node()，并传递给它合适的参数。alloc_pages_node()其实是一个包装函数，用于对请求的页面数量进行合理性检查。

```

-----
include/linux/gfp.h
67  static inline struct page * alloc_pages_node(int nid, unsigned int gfp_mask, unsigned
    int order)
68  {
69    if (unlikely(order >= MAX_ORDER))
70      return NULL;
71
72    return __alloc_pages(gfp_mask, order, NODE_DATA(nid)->node_zonelist + (gfp_mask &
    GFP_ZONEMASK));
73  }
-----
  
```

如你所见，如果请求页面数量大于允许的最大数量（MAX_ORDER），那么页面分配的请求会被终止。在函数 alloc_page()中页面数量（order）总被设置为 0，所以请求总能得到满足。MAX_ORDER 定义于文件 linux/mmzone.h 中，其值是 11，因此最多可以请求 2 048 个页面。

函数__alloc_pages()是真正的处理页面请求的函数。该函数定义在文件 mm/page_alloc.c 中，而且和前面讨论的内存管理区知识相关。

2. `__get_free_page()`和`__get_dma_pages()`

宏`__get_free_page()`是请求单页操作的简化版本。它和`alloc_page()`一样，在调用`__get_free_pages()`时传入 0 作为请求的页面数量（函数`__get_free_pages()`用来执行多页面请求），图 4-3 描述了这些函数的调用层次关系。

```
-----
include/linux/gfp.h
83 #define __get_free_page(gfp_mask) \
84     __get_free_pages((gfp_mask), 0)
-----
```

宏`__get_dma_pages`指定从 `ZONE_DMA` 请求的页面，这是通过把 `GFP_DMA` 标志加到页标志掩码实现的。`ZONE_DMA` 指向的内存管理区是专为 DMA 访问所预留的。

```
-----
include/linux/gfp.h
86 #define __get_dma_pages(gfp_mask, order) \
87     __get_free_pages((gfp_mask) | GFP_DMA, (order))
-----
```

4.3.2 释放页面的函数

为了释放页面，内核提供了多个例程，包括两个宏和两个包装器函数。图 4-4 描述这些页释放例程之间的调用层次。依照它们的参数类型我们将这些函数分为两组。第一组函数为`__free_page()`和`__free_pages()`，其参数是指向待释放页的页描述符指针；第二组函数为`free_page()`和`free_pages()`，其参数为待释放页面的首地址。

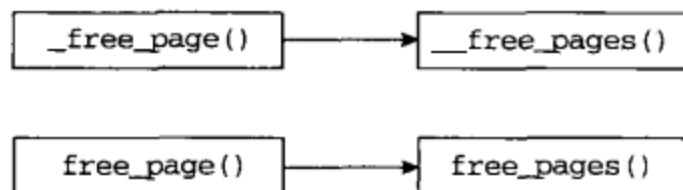


图 4-4 `*free_page*`()的调用层次

宏`__free_page()`和`free_page()`用于释放单页面。在调用页面释放函数时，将需释放的页面数量参数置为 0，函数`__free_pages()`和`free_pages()`的定义分别如下：

```
-----
include/linux/gfp.h
94 #define __free_page(page) __free_pages((page), 0)
95 #define free_page(addr) free_pages((addr), 0)
-----
```

函数`free_pages()`最终调用函数`__free_pages_bulk()`，后者是 Linux 伙伴系统的页面释放函数。我们将在下一节中详细讨论伙伴系统。

4.3.3 伙伴系统

每当页面被分配和回收时，系统都要遇到名为外部碎片（external fragmentation）的内存碎

片问题。这是由于可用页面散布于整个内存空间中，即使系统可用页面总数足够多，但也无法分配大块连续页面。这就是说，可用页面会被一个或多个不连续的不可用页面拆开。减少外部碎片的途径很多，Linux 使用伙伴系统 (buddy system) 这样的内存管理算法的实现来达到该目的。

伙伴系统把内存中的空闲块组成链表。每个链表都指向不同大小的内存块，虽然大小不同，但都是 2 的幂次方。至于系统中有多少个链表，则取决于具体实现。从最小的空闲块链表中分配页面，这样保证了较大的连续块可留给更大的内存请求。当分配的块被释放时，伙伴系统搜索与所释放块大小相等的可用空闲内存块，如果找到相邻的空闲块，那么就将它们合并成两倍于自身大小的一个新内存块。这些可合并的内存块（所释放的块和与其相邻的空闲块）就称为伙伴，所以也就有了伙伴系统一说。之所以这么做，是因为内核需要确保只要页面被释放，就能获得更大的可用内存块。

现在来看看 Linux 中实现伙伴系统的具体函数，其中页面分配函数是 `__alloc_pages()` (`mm/page_alloc.c`)，页面回收函数是 `__free_pages_bulk()`：

```
-----
mm/page_alloc.c
585 struct page * fastcall
586 __alloc_pages(unsigned int gfp_mask, unsigned int order,
587               struct zonelist *zonelist)
588 {
589     const int wait = gfp_mask & __GFP_WAIT;
590     unsigned long min;
591     struct zone **zones;
592     struct page *page;
593     struct reclaim_state reclaim_state;
594     struct task_struct *p = current;
595     int i;
596     int alloc_type;
597     int do_retry;
598
599     might_sleep_if(wait);
600
601     zones = zonelist->zones;
602     if (zones[0] == NULL) /* no zones in the zonelist */
603         return NULL;
604
605     alloc_type = zone_idx(zones[0]);
606     ...
607     for (i = 0; zones[i] != NULL; i++) {
608         struct zone *z = zones[i];
609
610         min = (1<<order) + z->pages_low;
611         ...
612         if (rt_task(p))
613             min -= z->pages_low >> 1;
614
615         if (z->free_pages >= min ||
616             (!wait && z->free_pages >= z->pages_high)) {
```



```

622     page = buffered_rmqueue(z, order, gfp_mask);
623     if (page) {
624         zone_statistics(zonelist, z);
625         goto got_pg;
626     }
627 }
628 }
629
630 /* we're somewhat low on memory, failed to find what we needed */
631 for (i = 0; zones[i] != NULL; i++)
632     wakeup_kswapd(zones[i]);
633
634 /* Go through the zonelist again, taking __GFP_HIGH into account */
635 for (i = 0; zones[i] != NULL; i++) {
636     struct zone *z = zones[i];
637
638     min = (1<<order) + z->protection[alloc_type];
639
640     if (gfp_mask & __GFP_HIGH)
641         min -= z->pages_low >> 2;
642     if (rt_task(p))
643         min -= z->pages_low >> 1;
644
645     if (z->free_pages >= min ||
646         (!wait && z->free_pages >= z->pages_high)) {
647         page = buffered_rmqueue(z, order, gfp_mask);
648         if (page) {
649             zone_statistics(zonelist, z);
650             goto got_pg;
651         }
652     }
653 }
...
720 nopage:
721     if (!(gfp_mask & __GFP_NOWARN) && printk_ratelimit()) {
722         printk(KERN_WARNING "%s: page allocation failure."
723             " order:%d, mode:0x%x\n",
724             p->comm, order, gfp_mask);
725         dump_stack();
726     }
727     return NULL;
728 got_pg:
729     kernel_map_pages(page, 1 << order, 1);
730     return page;
731 }

```

Linux 伙伴系统按内存管理区管理，这就意味可用页面链表也要按区分别维护，因此，可用页面的搜索可能发生在三个内存管理区中。

第 586 行：整数值 `gfp_mask` 让函数 `__alloc_pages()` 的调用者指定搜索页面的方式（操作修饰符）。该变量可能的取值定义在 `/linux/gfp.h` 中，表 4-2 列举了取值的详细情况。

表4-2 页面分配方式（操作修饰符gfp_mask）

标 志	描 述
__GFP_WAIT	允许内核阻塞进程等待页面。比如page_alloc.c中第537行使用到了它
__GFP_COLD	请求缓冲调清页面（cache cold page）
__GFP_HIGH	页面可从应急内存池中找到
__GFP_IO	页面可用于执行I/O传输
__GFP_FS	允许页面调用低级FS操作
__GFP_NOWARN	页面分配失败后，分配函数就发送一个失败警告，如果设置该修饰符，那么禁止警告。其用法见page_alloc.c中第665~666行
__GFP_REPEAT	请求失败后尝试再分配
__GFP_NORETRY	请求失败后不再重复
__GFP_DMA	页面处于ZONE_DMA[X]
__GFP_HIGHMEM	页面处于ZONE_HIGHMEM[X]

表 4-3 提供了指向内存管理区链表（zonelist）的指针，内存管理区链表的修饰符来自 gfp_mask。

表4-3 内存管理区链表

标 识	描 述
GFP_USER	表示不应该在内核内存中分配内存
GFP_KERNEL	表示将从内核内存中分配内存
GFP_ATOMIC	中断处理程序中调用kmalloc需设置该标志，它标识内存分配时调用者不可睡眠
GFP_DMA	表示将从ZONE_DMA区分配内存

第 599 行：函数 `might_sleep_if()` 获取 `wait` 变量的值（`wait` 是 `gfp_mask` 和 `__GFP_WAIT` 的逻辑与所得）。如果未设置 `__GFP_WAIT`，则 `wait` 值为 0；如果设置 `__GFP_WAIT`，则该值为 1。如果编译内核时检查到自旋锁内睡眠（Sleep-inside-spinlock）被启用（在内核编译选项的 Kernel Hacking 菜单下），那么该函数将允许内核将当前进程阻塞指定的超时时间。

第 608~628 行：在该代码段中，我们遍历内存管理区描述符链表，搜索一个可以为请求提供足够空闲页面的内存管理区。如果区中的空闲页面数可满足要求或如果请求进程允许等待，而且区中空闲页面数大于或等于区的上限值，那么就要调用函数 `buffered_rmqueue()`。

函数 `buffered_rmqueue()` 有三个参数：可用页面的内存管理区描述符，请求的页面数，请求页面的动作修饰符。

第 631~632 行：如果进入该代码段，说明我们将无法分配页面，因为可用页面数已经处于很低的状态，这时要做的是尝试回收页面来满足需要。函数 `wakeup_kswapd()` 实现这一功能，它会给内存管理区补充适量的页面，并适当更新区描述符。

第 635~653 行：当我们在前面代码段中尝试补充页面后，要重新进入内存管理区链表，再次搜寻满足请求的足够多的空闲页面。

第 720~727 行：在确定无页面可用的情况下才进入该代码段。如果未设置修饰符 `GFP_NOWARN`，那么就打印一个页面分配失败的警告，警告信息包括调用当前进程的命令名、所请求的页面大小、应用到这个请求的 `gfp_mask` 等信息。然后函数返回 `NULL`。

第 728~730 行：找到所要的页面后进入该代码段。函数返回一个页描述符的地址，如果请求超过一页，则返回所分配的首页描述符的地址。

当内存块被回收时，伙伴系统检查内存管理区中是否存在与该内存块同样大小的伙伴。如果存在，那么伙伴系统将它们合并成较大的内存块。函数 `__free_pages_bulk()` 便是执行该操作的。我们看看它如何工作：

```
-----
mm/page_alloc.c
178 static inline void __free_pages_bulk (struct page *page, struct page *base,
179     struct zone *zone, struct free_area *area, unsigned long mask,
180     unsigned int order)
181 {
182     unsigned long page_idx, index;
183
184     if (order)
185         destroy_compound_page(page, order);
186     page_idx = page - base;
187     if (page_idx & ~mask)
188         BUG();
189     index = page_idx >> (1 + order);
190
191     zone->free_pages -= mask;
192     while (mask + (1 << (MAX_ORDER-1))) {
193         struct page *buddy1, *buddy2;
194
195         BUG_ON(area >= zone->free_area + MAX_ORDER);
196         if (!__test_and_change_bit(index, area->map))
197             ...
206         buddy1 = base + (page_idx ^ ~mask);
207         buddy2 = base + page_idx;
208         BUG_ON(bad_range(zone, buddy1));
209         BUG_ON(bad_range(zone, buddy2));
210         list_del(&buddy1->lru);
211         mask <=> 1;
212         area++;
213         index >>= 1;
214         page_idx &= mask;
215     }
216     list_add(&(base + page_idx)->lru, &area->free_list);
217 }
```

第 184~215 行：函数 `__free_pages_bulk()` 遍历所有空闲块链（内存块最大不可超过 2 的 `MAX_ORDER` 次幂）。查遍所有空闲块，如果已经到达块的最大尺寸或该函数找到最小的伙伴后，就会调用函数 `__test_and_change_bit()` 来测试看我们所释放块的伙伴是否已被分配。如果伙伴页已被分配，那么跳出循环；如果还没被分配，那么则查看是否可以找到更大的伙伴块与页面的释放块合并。

第 216 行：将空闲块插入到合适的空闲页面链中。

4.4 Slab 分配器

我们已经知道页是内存管理的基本单元。但是进程往往会以字节为单位请求内存，而非以页

为单位请求内存。为了支持这种小块内存请求（比如调用 `kmalloc()`），内核特别实现了 slab 分配器（slab allocator），slab 分配器是一种对所请求的页进行管理的内存管理层。

Slab 分配器为了减少内存分配、初始化、销毁和释放的代价，通常会维护经常使用的内存区的一个现成的缓存区。这个缓存区维护着分配的、初始化的以及准备部署的内存区。当请求进程不再需要内存区时，就会把释放的内存送回缓存区中。

实际上，slab 分配器由许多缓存组成，不同的缓存存储大小不同的内存区。缓存可以是专用的（specialized），也可能是通用的（general purpose），专用缓存存储保存特定对象的内存区，比如各种描述符，像进程描述符（`task_structs`）就存放在 slab 分配器维护的缓存中，该缓存存储的内存区的大小为 `task_struct` 结构体的大小 `sizeof(task_struct)`。同样，`inode` 和 `dentry` 数据结构也存放在缓存中。一般来讲，通用缓存是预定义大小的内存区组成，其大小可以是：32、64、128、256、512、1 024、2 048、4 096、8 192、16 384、32 768、65 536 和 131 072 字节^①。

如果执行命令 `cat /proc/slabinfo`，系统中现有的 slab 分配器缓存都显示出来。在输出结果的第一列，可看到很多内核数据结构的名称和一组 `size-*` 格式的表项。前者对应存放特定对象的专用缓存，后者对应存放大小固定的通用对象的通用缓存。

你可能已经注意到任何大小的通用缓存都有两项，其中一个是以（DMA）结束。之所以有两个类似项，是由于既可从 DMA 内存管理区请求内存区^②，也可以从普通内存管理区请求，所以 slab 分配器同时维护了上述两种内存管理区对应的缓存，从而方便满足各种内存请求。图 4-5 显示了 `/proc/slabinfo` 的输出信息，从中就可看到两种类型内存的缓存。

size-131072 (DMA)	0	0	131072	1	32	tunables	8	4	0	slabdata	0	0
size-131072	0	0	131072	1	32	tunables	8	4	0	slabdata	0	0
size-65536 (DMA)	0	0	65536	1	16	tunables	8	4	0	slabdata	0	0
size-65536	1	1	65536	1	16	tunables	8	4	0	slabdata	1	1
size-32768 (DMA)	0	0	32768	1	8	tunables	8	4	0	slabdata	0	0
size-32768	0	0	32768	1	8	tunables	8	4	0	slabdata	0	0
size-16384 (DMA)	0	0	16384	1	4	tunables	8	4	0	slabdata	0	0
size-16384	0	0	16384	1	4	tunables	8	4	0	slabdata	0	0
size-8192 (DMA)	0	0	8192	1	2	tunables	8	4	0	slabdata	0	0
size-8192	64	68	8192	1	2	tunables	8	4	0	slabdata	64	68
size-4096 (DMA)	0	0	4096	1	1	tunables	24	12	0	slabdata	0	0
size-4096	65	65	4096	1	1	tunables	24	12	0	slabdata	65	65
size-2048 (DMA)	0	0	2048	2	1	tunables	24	12	0	slabdata	0	0
size-2048	102	102	2048	2	1	tunables	24	12	0	slabdata	51	51
size-1024 (DMA)	0	0	1024	4	1	tunables	54	27	0	slabdata	0	0
size-1024	73	100	1024	4	1	tunables	54	27	0	slabdata	25	25
size-512 (DMA)	0	0	512	8	1	tunables	54	27	0	slabdata	0	0
size-512	288	288	512	8	1	tunables	54	27	0	slabdata	36	36
size-256 (DMA)	0	0	256	15	1	tunables	120	60	0	slabdata	0	0
size-256	149	165	256	15	1	tunables	120	60	0	slabdata	11	11
size-128 (DMA)	0	0	128	30	1	tunables	120	60	0	slabdata	0	0
size-128	4906	10290	128	30	1	tunables	120	60	0	slabdata	343	343
size-64 (DMA)	0	0	64	58	1	tunables	120	60	0	slabdata	0	0
size-64	1565	2323	64	58	1	tunables	120	60	0	slabdata	40	40

图 4-5 `cat /proc/slabinfo`

① 为了提高性能，所有通用缓存都是 L1 对齐的。

② 注意区别内存管理区（zone）和内存区（memory area）。——译者注

可以把缓存进一步划分成所谓 slab 的容器，每个 slab 都由一个或多个连续的页面组成，小内存区便从 slab 容器这里分配，这就是为什么我们说 slab 容纳的是对象。对象本身便是属于 slab 的页面中预定义大小的地址区间，图 4-6 显示了 slab 分配器的组织关系。

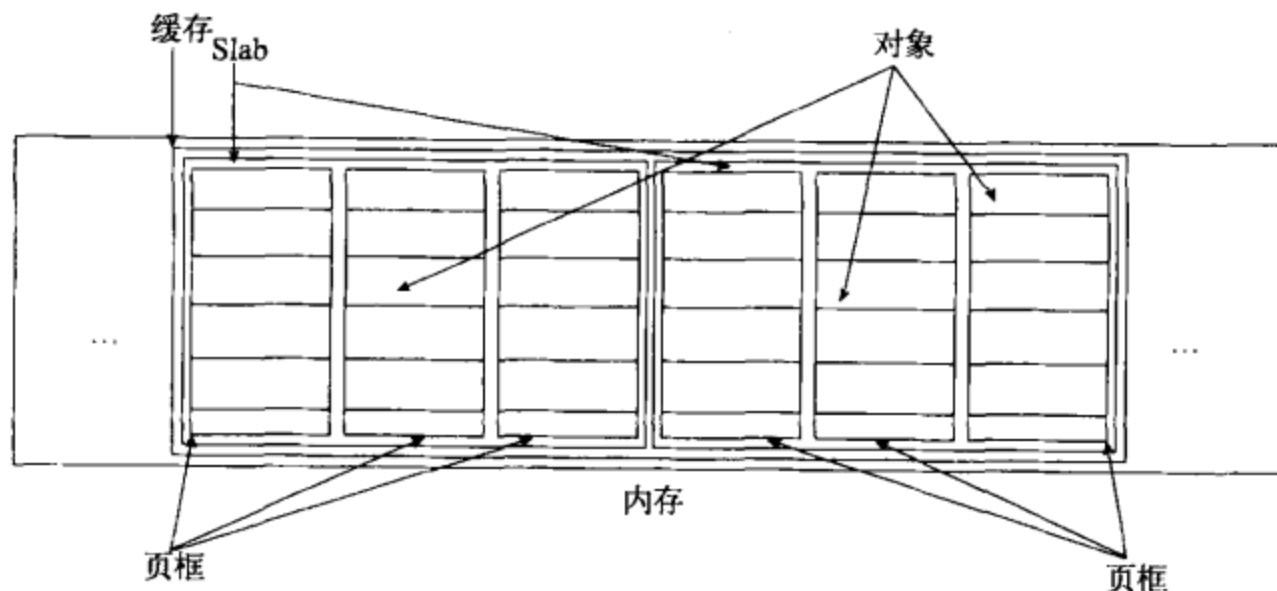


图 4-6 Slab 分配器组织关系图

Slab 分配器使用三个主要结构维护对象信息，它们分别是：称为 `kmem_cache` 的缓存描述符，称为 `cache_sizes` 的通用缓存描述符和称为 `slab` 的 slab 描述符。图 4-7 总结了所有这些描述符之间的逻辑关系。

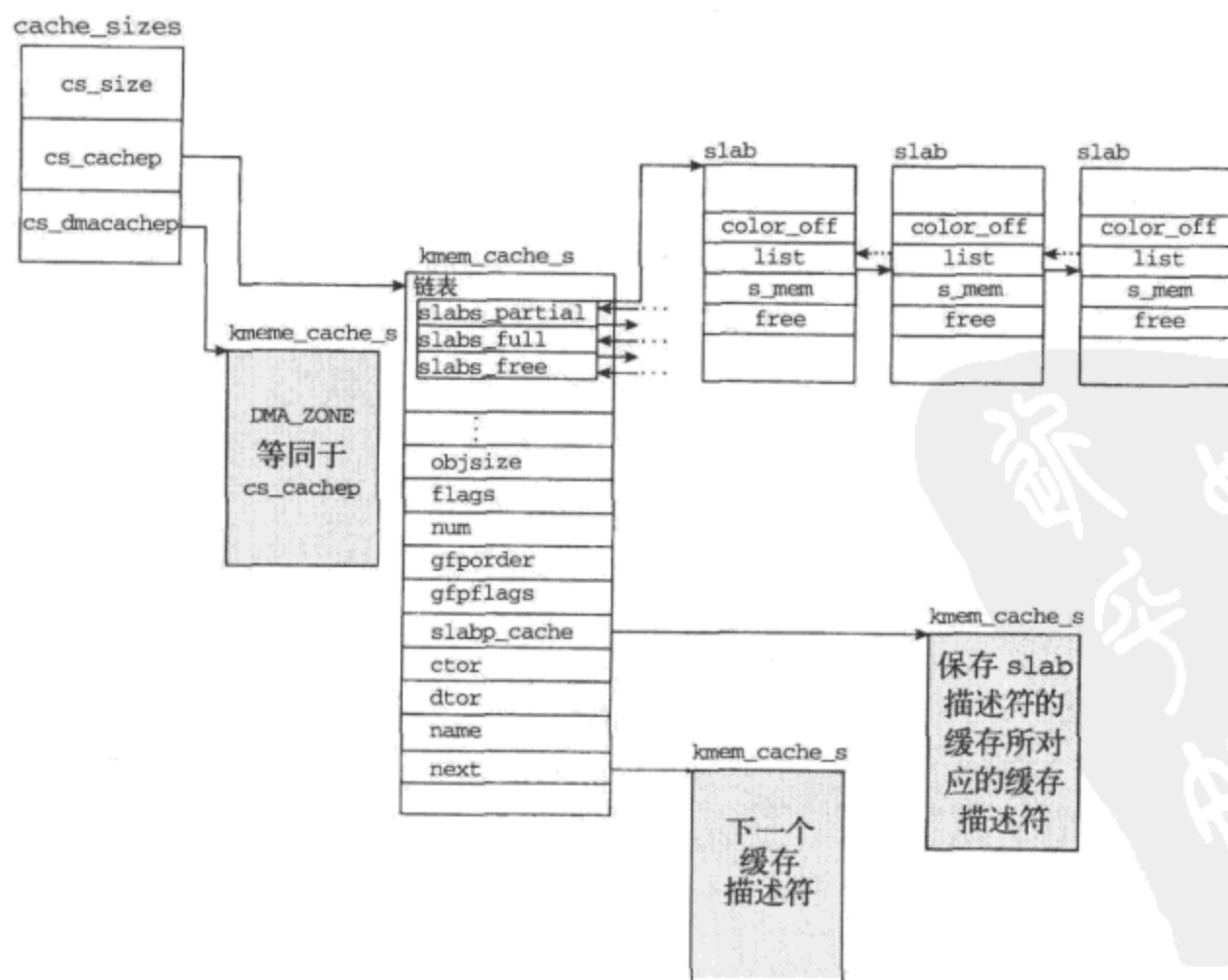


图 4-7 Slab 分配器结构

4.4.1 缓存描述符

每一个缓存都有一个类型为 `kmem_cache_s` 的缓存描述符，描述符中包含缓存的许多信息。这些信息中大多数值都是由函数 `kmem_cache_create()` (`mm/slab.c`) 在缓存创建时设置或计算获得的。我们会在后面的章节中讨论它们，这里我们首先来看看缓存描述符中的一些字段以及它们存储的信息。

```
-----
mm/slab.c
246 struct kmem_cache_s {
...
252 struct kmem_list3 lists;
...
254 unsigned int objsize;
255 unsigned int flags; /* constant flags */
256 unsigned int num; /* # of objs per slab */
...
263 unsigned int gfporder;
264
265 /* force GFP flags, e.g. GFP_DMA */
266 unsigned int gfpflags;
267
268 size_t color; /* cache coloring range */
269 unsigned int color_off; /* color offset */
270 unsigned int color_next; /* cache coloring */
271 kmem_cache_t *slabp_cache;
272 unsigned int dflags; /* dynamic flags */
273
273 /* constructor func */
274 void (*ctor)(void *, kmem_cache_t *, unsigned long);
275
276 /* de-constructor func */
277 void (*dtor)(void *, kmem_cache_t *, unsigned long);
278
279 /* 4) cache creation/removal */
280 const char *name;
281 struct list_head next;
282
...
301 };
-----
```

1. list

`list` 字段中包含三个链表头，其中每个链表头对应 `slab` 所处的三种状态之一：部分、完全和空闲。一个缓存可以有一个或多个 `slab` 处于上述任何一个状态中。缓存正是通过这个数据结构引用 `slab`。`list` 本身是由 `slab` 描述符的 `list` 字段维护着的双向链表。有关“`slab` 描述符”的内容将在本章后续部分陆续讲到。

```
-----
mm/slab.c
217 struct kmem_list3 {
218 struct list_head slabs_partial;
219 struct list_head slabs_full;
220 struct list_head slabs_free;
...

```



```

223 unsigned long next_reap;
224 struct array_cache *shared;
225 };

```

● lists.slabs_partial

lists.slabs_partial 是指 slab 中只有部分对象被分配的链表头，也就是说，如果 slab 处在“部分”状态下，表示它有一部分对象被分配，还有一部分空闲待用。

● lists.slabs_full

lists.slabs_full 是所有对象都已被分配的 slab 链表头，也就是说，slab 中没有可用对象。

● lists.slabs_free

lists.slabs_free 是指所有对象都空闲的 slab 链表头，也就是说，该 slab 中所有的对象都可用。

维护这些链表的目的是为了缩减寻找空闲对象的时间。当请求缓存中的对象时，内核将搜索部分 slab 链，如果部分 slab 链是空的，那么将搜索空闲 slab，如果空闲 slab 链表是空，那么将创建一个新的 slab。

● lists.next_reap

slab 存放分配给它们的页面，如果这些页并未使用，那最好将它们退回给主内存池。最后，缓存也会被收回。该字段存放的是下一个缓冲回收的时间，在调用函数 kmem_cache_create() (mm/slab.c) 创建缓存时设置该字段，并且每次调用函数 cache_reap() 时更新该字段。

2. objsize

objsize 字段存放缓存中对象的大小（以字节为单位），该值取决于缓存创建时请求的大小以及对齐的需要。

3. flag

flag 字段存放标志掩码，掩码描述缓存的固有特性。include/linux/slab.h 文件中定义了所有可能的标志。表 4-4 详细描述这些标志的含义。

表4-4 slab标志

flag 名称	描 述
SLAB_POISON	要求在创建slab时将a5a5a5a5作为测试数据写入slab中。该标志用于验证已初始化的内存
SLAB_NO_REAP	当出现内存不足时，内存管理器将开始回收没有使用的内存。设置该标志将保证该缓存不会在此情况下被自动回收
SLAB_HWCACHE_ALIGN	要求对象与处理器的硬件缓存线（cacheline）对齐，以便减少内存访问周期，从而提高系统访问性能
SLAB_CACHE_DMA	表示将使用DMA内存。当请求新页面时，GFP_DMA标志会传给伙伴系统
SLAB_PANIC	表示不管因为什么原因，如果函数kmem_cache_create()失败，则调用panic

4. num

num 字段存放缓存中每个 slab 所包含的对象数目。它是在缓存创建（也使用 kmem_cache_create()）时，根据 gfporder 的值（详见下个字段）、需创建对象的大小以及对齐要求确定的。

5. gfporder

`gfporder` 是缓存中每个 slab 所占连续页面数的幂（基数为 2）。值默认为 0，缓存创建时调用函数 `kmem_cache_create()` 时设置该值。

6. gfpflags

`gfpflags` 标志指定缓存中为 slab 分配的页面类型。这些类型由内存区请求的标志决定，比如，如果是 DMA 将使用的内存，那么 `gfpflags` 字段将被设置为 `GFP_DMA`，而且该字段将在请求页面时作为参数被传入。

7. slabp_cache

slab 描述符可以存放在缓存自身内，也可存放在外部其他缓存中。如果这个缓存中 slab 的描述符存放在自身以外的缓存中，那么 `slabp_cache` 字段保存指向缓存所对应的缓存描述符的指针，这些缓存存储 Slab 描述符对象。请看 4.4.3 节，了解更多 slab 描述符存放的信息。

8. ctor

如果存在构造函数，则 `ctor` 字段指向与缓存相关的构造函数的指针^①。

9. dtor

与 `ctor` 字段类似，如果存在析构函数，则 `dtor` 字段指向与缓存相关的析构函数的指针。

构造函数和析构函数都在缓存创建时定义，而且作为参数传递给函数 `kmem_cache_create()`。

10. name

`name` 字段存放可读性好的字符串的名称，可以在打开 `/proc/slabinfo` 时显示。比如对于存放文件指针的缓存，则该字段的值为 `filp`。调用 `cat /proc/slabinfo` 能更好地理解这一点。`name` 字段存放的名称要唯一。在创建时为 Slab 指定的名称，要与链表中所有其他 slab 的名称进行比较，看是否有重名。如果发现有相同名称的 slab 已存在，那么 slab 创建失败。

11. next

`next` 字段是指向缓存描述符单链表中下一个缓存描述符的指针。

4.4.2 通用缓存描述符

前面我们提到，通用缓存总是成对出现，其中存放预定大小的对象。一个缓存从 DMA 内存区分配对象，另一个缓存从普通内存区中分配。如果你还记得前面提到的内存管理区，就应该知道 DMA 缓存是在 `ZONE_DMA` 区中，而标准缓存则来自 `ZONE_NORMAL` 区。`struct cache_sizes` 存放通用缓存大小的所有信息。

```
-----
include/linux/slab.h
69 struct cache_sizes {
70     size_t    cs_size;
71     kmem_cache_t *cs_cachep;
72     kmem_cache_t *cs_dmacachep;
73 };
-----
```

^① 对于熟悉面向对象程序开发的读者来说，构造函数和析构函数都不陌生。缓存描述符的 `ctor` 字段让构造函数在每次创建新缓存描述符时被调用。类似地，`dtor` 字段存放指向每次销毁缓存描述符时调用的析构函数的指针。

1. cs_size

cs_size 字段存放该缓存中所包含的内存对象大小。

2. cs_cachep

cs_cachep 字段存放指向普通内存缓存描述符的指针，这种缓存存放的对象分配自 ZONE_NORMAL 内存区。

3. cs_dmacachep

cs_cachep 字段存放指向 DMA 内存缓存描述符的指针，DMA 内存缓存存放的对象分配自 ZONE_DMA。

现在有个问题，“缓存描述符本身存放在哪里？”slab 分配器有一个缓存专门预留来解决该问题。cache_cache 缓存便是用来存放缓存描述符对象的。slab 缓存是在系统自举时被静态初始化，以确保缓存描述符存储空间可用。

4.4.3 Slab 描述符

在缓存中的每个 slab 都包含针对该 slab 特定信息的描述符，我们已经提到，缓存描述符存放在叫做 cache_cache 的专用缓存中。而 slab 描述符可依次存放在两个地方：在 slab 本身中（特指在第一个页面中），或者在外部的第一个“通用”缓存中，当然这个缓存要足够大以容纳 slab 描述符。具体存放位置要根据在缓存创建时满足对象对齐要求后剩余空闲空间综合考虑。剩余空间要在缓存创建时确定。

下面来看看 slab 描述符的一些字段。

```
-----
mm/slab.c
173 struct slab {
174     struct list_head list;
175     unsigned long coloroff;
176     void *s_mem; /* including color offset */
177     unsigned int inuse; /* num of objs active in slab */
178     kmem_bufctl_t free;
179 };
-----
```

1. list

如果你回想一下关于缓存描述符的讨论，就可知道 slab 可以处于三种状态之一：free、partial 或 full。缓存描述符中存放处于全部三种状态的 slab 链表，每个状态对应一个链表，所有状态的 slab 都通过 list 字段保存在双向链表中。

2. s_mem

s_mem 字段存放的指针指向 slab 中的第一个对象。

3. inuse

inuse 值记录 slab 中含有对象的数量。对于满或部分满的 slab，该值为正值；对于空闲 slab，该值为 0。

4. free

free 字段存放 slab 对象数组的索引值，该字段所包含数组项的索引值，这个数组项表示 slab

中第一个可用对象。kmem_bufctl_t 数据类型链接处于同一 slab 中的所有对象，该数据类型仅仅是一个无符号整数，定义于文件 include/asm/types.h 中。这些数据类型组成了一个数组，且无论 slab 描述符存放在 slab 内部或外部，这个数组总是正好存放在 slab 描述符之后。要知详情，请察看内联函数 slab_bufctl()，该函数会返回对象数组。

```
-----
mm/slab.c
1614 static inline kmem_bufctl_t *slab_bufctl(struct slab *slabp)
1615 {
1616     return (kmem_bufctl_t *) (slabp+1);
1617 }
-----
```

函数 slab_bufctl() 接收一个指向 slab 描述符的指针，返回紧跟在 slab 描述符后的内存地址指针。

当缓存被初始化时，slab->free 字段被设置成 0（因为所有对象都是空闲的，因此应该返回第一个对象），而且 kmem_bufctl_t 数组中的每一项都设置为数组中下一个元素的索引值，这就是说第 0 个元素存放的值为 1，第一个元素存放的值为 2，以此类推。数组的最后一个元素存放值 BUFCTL_END，表示数组已经到头了。

图 4-8 描述了当 slab 描述符存放在自身内部时，slab 描述符、bufctl 数组以及 slab 对象的布局。表 4-5 显示了当 slab 处于三中可能状态时，slab 描述符字段可能的取值。

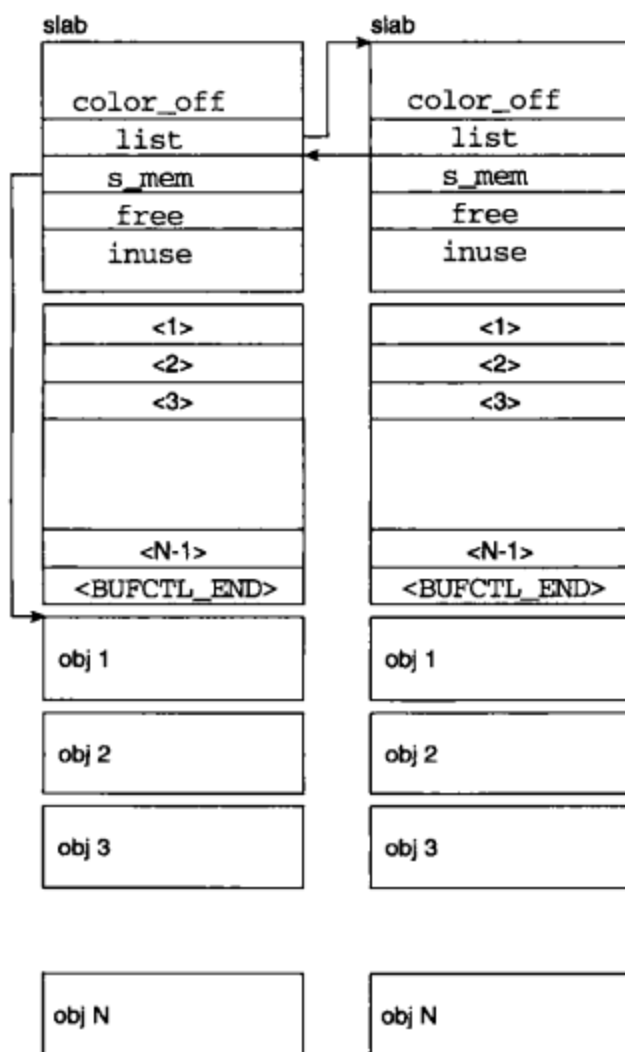


图 4-8 Slab 描述符和 bufctl

表4-5 Slab状态和描述符字段的值

	Free	Partial	Full
slab->inuse	0	X	N
slab->free	0	X	N

N = slab 中的对象数。

X = 某一可变的正数。

4.5 Slab 分配器的生命周期

现在，我们在内核运行的整个生命期范围内，观察缓存和 slab 分配器是如何交互的。内核需要确保某些结构正好能支持进程对内存区的请求，某些结构能支持动态可加载模块对专用缓存的动态创建。

因此，有一些全局结构对 slab 分配器具有关键作用。其中一些我们在前面章节未作介绍，这里我们来看看这些全局变量。

4.5.1 与 Slab 分配器有关的全局变量

有下面一组与 slab 分配器相关的全局变量。

- ❑ **cache_cache**: 它是一个特殊缓存的缓存描述符，其特殊性体现在该缓存存放的是其他所有缓存描述符。这个缓存的名称是 `kmem_cache`（在 `/proc/slabs/` 中的名字）。该缓存描述符是唯一一个静态分配的缓存描述符。
- ❑ **cache_chain**: 用作指向缓存描述符链表的指针的链表元素。
- ❑ **cache_chain_sem**: 对 `cache_chain` 进行控制访问的信号量^①。每次向链表中添加元素（新缓存描述符）时，分别调用 `down()` 和 `up()` 函数获取和释放信号量。
- ❑ **malloc_sizes[]**: 该数组存放 DMA 区的缓存描述符和与通用缓存对应的非 DMA 区的缓存描述符。

在 slab 分配器初始化之前，这些结构已经存在，让我们看看它们的创建过程：

```
-----
mm/slab.c
486 static kmem_cache_t cache_cache = {
487     .lists    = LIST3_INIT(cache_cache.lists),
488     .batchcount = 1,
489     .limit    = BOOT_CPUCACHE_ENTRIES,
490     .objsize  = sizeof(kmem_cache_t),
491     .flags    = SLAB_NO_REAP,
492     .spinlock = SPIN_LOCK_UNLOCKED,
493     .color_off = L1_CACHE_BYTES,
494     .name     = "kmem_cache",
495 };
496
497 /* Guard access to the cache-chain. */
```

① 信号量的详细信息将在第9章讨论。

```

498 static struct semaphore cache_chain_sem;
499
500 struct list_head cache_chain;
-----

```

cache_cache 缓存描述符中若设置了 SLAB_NO_REAP 标志, 那么即使系统内存紧张, 该缓存也会在内核运行期间驻留内存。注意, cache_chain 信号量仅被定义, 并未初始化。初始化是在系统初始化时由函数 kmem_cache_init() 完成的。我们来看看该函数的实现细节:

```

-----
mm/slab.c
462 struct cache_sizes malloc_sizes[] = {
463 #define CACHE(x) { .cs_size = (x) },
464 #include <linux/kmalloc_sizes.h>
465 { 0, }
466 #undef CACHE
467 };
-----

```

这段代码初始化 malloc_sizes[] 数组, 并按照在 include/linux/kmalloc_sizes.h 文件中定义的值设置 cs_size 字段。正如我们以前提到的, 缓存大小的取值范围是 32 字节~131 072 字节, 具体值取决于内核配置^①。

依靠这些全局变量, 内核通过调用文件 init/main.c^② 中的函数 kmem_cache_init() 来继续初始化 slab 分配器。该函数负责初始化缓存链表、信号量、通用缓存、kmem_cache 缓存, 本质上也就是初始化管理 slab 的 slab 分配器用到的所有全局变量。之后, 专用缓存方可被创建。创建缓存的函数为 kmem_cache_create()。

4.5.2 创建缓存

创建缓存包含三个步骤:

- (1) 描述符的分配和初始化;
- (2) 计算 slab 着色和对象大小;
- (3) 在 cache_chain 链表中添加缓存。

通用缓存是由函数 kmem_cache_init() (mm/slab.c) 在系统初始化时创建的, 专用缓存则由函数 kmem_cache_create() 创建的。

我们分别来看看这两个函数。

1. kmem_cache_init()

该函数创建 cache_chain 和通用缓存, 它是在初始化过程中被调用的。注意该函数名前有一个 _init 前缀, 在第 2 章解释过, _init 修饰的函数载入内存后会在自举和初始化过程结束后被销毁。

① 内核选项中有一些额外的配置选项可支持超过 131 072 字节的通用缓存。更多信息可参阅文件 include/linux/kmalloc_sizes.h。

② 第 9 章将会把系统自启动上电开始的初始化过程——道来。我们将看到函数 kmem_cache_init() 如何在自举过程中发挥作用。


```

-----
mm/slab.c
659 void __init kmem_cache_init(void)
660 {
661     size_t left_over;
662     struct cache_sizes *sizes;
663     struct cache_names *names;
664     ...
669     if (num_physpages > (32 << 20) >> PAGE_SHIFT)
670         slab_break_gfp_order = BREAK_GFP_ORDER_HI;
671
672
-----

```

第 661~663 行：变量 `sizes` 和 `names` 是由 `kmalloc` 分配的数组（大小为几何分布的通用缓存）的头。此刻这些数组被置于 `_init` 数据区。注意 `kmalloc()` 此时还并不存在。函数 `kmalloc()` 使用我们刚刚建立的 `malloc_sizes` 数组。到此为止，我们所拥有的只有静态分配的 `cache_cache` 描述符。

第 669~670 行：这个代码段确定一个 slab 使用多少页。一个 slab 中的页面数完全取决于系统有多少可用内存。在 x86 和 PPC 体系机构中，变量 `PAGE_SHIFT` (`include/asm/page.h`) 值为 12，所以我们要确认 `num_physpages` 的值是否大于 8 K。如果超过的话说明我们的机器有超过 32 M 的内存，这时每个 slab 用 `BREAK_GFP_ORDER_HI` 个页；否则为每个 slab 分配 1 页。

```

-----
mm/slab.c
690     init_MUTEX(&cache_chain_sem);
691     INIT_LIST_HEAD(&cache_chain);
692     list_add(&cache_cache.next, &cache_chain);
693     cache_cache.array[smp_processor_id()] = &initarray_cache.cache;
694
695     cache_estimate(0, cache_cache.objsize, 0,
696         &left_over, &cache_cache.num);
697     if (!cache_cache.num)
698         BUG();
699
700
-----

```

第 690 行：该行初始化 `cache_chain` 所存放的信号量 `cache_chain_sem`。

第 691 行：初始化 `cache_chain` 链表，所有的缓存描述符都存放在该链表中。

第 692 行：向 `cache_chain` 链表中加入 `cache_cache` 描述符。

第 693 行：创建“每个 CPU”的缓存。其细节已经超出了本书范围。

第 695~698 行：该代码段执行完整性检查，也就是检查在 `cache_cache` 中是否至少分配了一个缓存描述符。而且该代码段还设置了 `cache_cache` 描述符的 `num` 字段，并计算还有多少剩余空间。这主要为了 slab 着色。slab 着色 (Slab Coloring) 是一种内核用于降低缓存对齐所带来的性能下降的方法。

```

-----
mm/slab.c
705  sizes = malloc_sizes;
706  names = cache_names;
707
708  while (sizes->cs_size) {
...
714      sizes->cs_cachep = kmem_cache_create(
715          names->name, sizes->cs_size,
716          0, SLAB_HWCACHE_ALIGN, NULL, NULL);
717      if (!sizes->cs_cachep)
718          BUG();
719
...
725
726      sizes->cs_dmacachep = kmem_cache_create(
727          names->name_dma, sizes->cs_size,
728          0, SLAB_CACHE_DMA|SLAB_HWCACHE_ALIGN, NULL, NULL);
729      if (!sizes->cs_dmacachep)
730          BUG();
731
732      sizes++;
733      names++;
734  }
-----

```

第 708 行：该行代码验证我们是否已经到达 `sizes` 数组的末尾，`sizes` 数组最后一个元素总被设置成 0，因此当我们到达数组最后一个元素时，判断条件就为真。

第 714~718 行：为常规内存分配请求创建下一个 `kmalloc` 缓存，并且验证该缓存是否为空。参见“`kmem_cache_create()`”部分。

第 726~730 行：这一代码段为 DMA 内存分配创建缓存。

第 732~733 行：找到 `sizes` 和 `names` 数组中的下一个元素。

函数 `kmem_cache_init()` 的剩余代码处理“用 `kmalloc` 分配的数据替换临时自举数据”的过程。我们省略介绍这一部分是因为它与实际的缓存描述符初始化没有直接关系。

2. `kmem_cache_create()`

当通用缓存所提供的内存区不能满足需要时，会调用该函数来创建专用缓存。创建专用缓存的步骤与创建通用缓存不大相同，依次包括：创建、分配、初始化缓存描述符、对齐对象、对齐 slab 描述符、添加缓存到缓存链表中。该函数没有 `__init` 前缀，因为它被调用后，可以使用永久内存。

```

-----
mm/slab.c
1027  kmem_cache_t *
1028  kmem_cache_create (const char *name, size_t size, size_t offset,
1029      unsigned long flags, void (*ctor)(void*, kmem_cache_t *, unsigned long),
1030      void (*dtor)(void*, kmem_cache_t *, unsigned long))
1031  {
1032      const char *func_nm = KERN_ERR "kmem_create: ";
1033      size_t left_over, align, slab_size;
1034      kmem_cache_t *cachep = NULL;
...
-----

```

我们来看看函数 `kmem_cache_create()` 的参数。

- name

用于标识缓存的名字。它存放在缓存描述符的 `name` 字段中，可以在文件 `/proc/slabinfo` 中看到这个名字。

- size

该参数指定缓存中包含的对象大小（以字节为单位），该值存放在缓存描述符的 `objsize` 字段中。

- offset

该值决定对象在一个页面上的位置。

- flags

`flags` 参数与 `slab` 相关。请参考表 4-4 对缓存描述符中 `flag` 字段和可能取值的描述。

- ctor和dtor

`ctor` 和 `dtor` 字段分别指向这个内存区中创建和销毁对象时调用的构造函数和析构函数。该函数执行大范围的调试与完整性检查，这里我们并未涉及。更详细的功能参见代码：

```
-----
mm/slab.c
1079  /* Get cache's description obj. */
1080  cachep = (kmem_cache_t *) kmem_cache_alloc(&cache_cache, SLAB_KERNEL);
1081  if (!cachep)
1082      goto opps;
1083  memset(cachep, 0, sizeof(kmem_cache_t));
1084
...
1144  do {
1145      unsigned int break_flag = 0;
1146      cal_wastage:
1147      cache_estimate(cachep->gfporder, size, flags,
1148                  &left_over, &cachep->num);
...
1174  } while (1);
1175
1176  if (!cachep->num) {
1177      printk("kmem_cache_create: couldn't create cache %s.\n", name);
1178      kmem_cache_free(&cache_cache, cachep);
1179      cachep = NULL;
1180      goto opps;
1181  }
-----
```

第 1079~1084 行：缓存描述符在此分配。此后的代码负责将 `slab` 中对象对齐。这里先不讨论对齐问题。

第 1144~1174 行：这段代码段决定缓存中对象的数量。大部分工作由函数 `cache_estimate()` 完成。计算出的对象数量最终会被存放在缓存描述符的 `num` 字段中。

mm/slab.c

```
...
1201  cachep->flags = flags;
1202  cachep->gfpflags = 0;
1203  if (flags & SLAB_CACHE_DMA)
1204    cachep->gfpflags |= GFP_DMA;
1205  spin_lock_init(&cachep->spinlock);
1206  cachep->objsize = size;
1207  /* NUMA */
1208  INIT_LIST_HEAD(&cachep->lists.slabs_full);
1209  INIT_LIST_HEAD(&cachep->lists.slabs_partial);
1210  INIT_LIST_HEAD(&cachep->lists.slabs_free);
1211
1212  if (flags & CFLGS_OFF_SLAB)
1213    cachep->slabp_cache = kmem_find_general_cachep(slab_size, 0);
1214  cachep->ctor = ctor;
1215  cachep->dtor = dtor;
1216  cachep->name = name;
1217
...
1242
1243  cachep->lists.next_reap = jiffies + REAPTIMEOUT_LIST3 +
1244    ((unsigned long)cachep)%REAPTIMEOUT_LIST3;
1245
1246  /* Need the semaphore to access the chain. */
1247  down(&cache_chain_sem);
1248  {
1249    struct list_head *p;
1250    mm_segment_t old_fs;
1251
1252    old_fs = get_fs();
1253    set_fs(KERNEL_DS);
1254    list_for_each(p, &cache_chain) {
1255      kmem_cache_t *pc = list_entry(p, kmem_cache_t, next);
1256      char tmp;
...
1265      if (!strcmp(pc->name, name)) {
1266        printk("kmem_cache_create: duplicate cache %s\n", name);
1267        up(&cache_chain_sem);
1268        BUG();
1269      }
1270    }
1271    set_fs(old_fs);
1272  }
1273
1274  /* cache setup completed, link it into the list */
1275  list_add(&cachep->next, &cache_chain);
1276  up(&cache_chain_sem);
1277  opps:
1278  return cachep;
1279 }
```

在此之前，slab 已经与硬件缓存对齐并进行了着色。slab 描述符中的 color 和 color_off 字段已被填充。

第 1200~1217 行：该代码段初始化缓存描述符的各个字段，与我们在 kmem_cache_init() 函数中看到的类似。

第 1243~1244 行：设置下一次回收缓存的时间。

第 1247~1276 行：初始化缓存描述符，并计算和存放与缓存相关的所有信息。现在我们将新的缓存描述符加入 cache_chain 链表中。

4.5.3 创建 slab 与 cache_grow()

当缓存刚被创建时，其中没有 slab。事实上，只有当一个对象发出请求的确需要 slab 时才真正分配 slab。这种情况出现在缓存描述符的 list.slabs_partial 和 lists.slabs_free 字段都为空时。此时我们并不关心对内存的请求如何转化成对特定缓存中对象的请求，而是假定这种转化已经发生，把关注点转移到 slab 分配器内的具体实现上。

函数 cache_grow() 创建 slab 并将其放在在缓存中。当我们创建 slab 时，不但分配和初始化其描述符，而且还需要分配物理内存，为此，我们需要与伙伴系统交互以请求页面。这个工作是由函数 kmem_getpages() (mm/slab.c) 完成的。

cache_grow()

函数 cache_grow() 用来在缓存中增加一个 slab，当缓存中没有空闲对象可用时就调用该函数。这种情况发生在 lists.slabs_partial 和 lists.slabs_free 都为空时。

```
-----
mm/slab.c
1546 static int cache_grow (kmem_cache_t * cachep, int flags)
1547 {
...
-----
```

传给该函数的参数有下面两个。

❑ **cachep**：它是需要扩充的缓存对应的缓存描述符。

❑ **flags**：这些标志将用于创建 slab。

```
-----
mm/slab.c
1572 check_irq_off();
1573 spin_lock(&cachep->spinlock);
...
1581
1582 spin_unlock(&cachep->spinlock);
1583
1584 if (local_flags & __GFP_WAIT)
1585     local_irq_enable();
-----
```

第 1572~1573 行：禁用中断且锁定描述符，为操作缓存描述符的字段做准备。

第 1582~1585 行：解锁缓存描述符且重新启用中断。

```

-----
mm/slab.c
...
1597  if (!(objp = kmem_getpages(cachep, flags)))
1598      goto failed;
1599
1600  /* Get slab management. */
1601  if (!(slabp = alloc_slabmgmt(cachep, objp, offset, local_flags)))
1602      goto opps1;
...
1605  i = 1 << cachep->gfporder;
1606  page = virt_to_page(objp);
1607  do {
1608      SET_PAGE_CACHE(page, cachep);
1609      SET_PAGE_SLAB(page, slabp);
1610      SetPageSlab(page);
1611      inc_page_state(nr_slab);
1612      page++;
1613  } while (--i);
1614
1615  cache_init_objs(cachep, slabp, ctor_flags);
-----

```

第 1597~1598 行：与伙伴系统交互以获取页面存放 slab。

第 1601~1602 行：把 slab 描述符存放在它该存放的地方。回想一下可以把 slab 描述符存放在 slab 自身或者第一个通用缓存中。

第 1605~1613 行：需要把页面与缓存和 slab 描述符关联。

第 1615 行：初始化 slab 中的所有对象。

```

-----
mm/slab.c
1616  if (local_flags & __GFP_WAIT)
1617      local_irq_disable();
1618  check_irq_off();
1619  spin_lock(&cachep->spinlock);
1620
1621  /* Make slab active. */
1622  list_add_tail(&slabp->list, &(list3_data(cachep)->slabs_free));
1623  STATS_INC_GROWN(cachep);
1624  list3_data(cachep)->free_objects += cachep->num;
1625  spin_unlock(&cachep->spinlock);
1626  return 1;
1627 opps1:
1628  kmem_freepages(cachep, objp);
1629 failed:
1630  if (local_flags & __GFP_WAIT)
1631      local_irq_disable();
1632  return 0;
1633 }
-----

```

第 1616~1619 行：由于我们打算访问并且改变描述符字段，所以需要禁用中断并锁定数据。

第 1622~1624 行：添加新的 slab 描述符到缓存描述符的 `lists.slabs_free` 字段中。更新记录缓存大小的统计信息。

第 1625~1626 行：解开自旋锁并且成功返回。

第 1627~1628 行：该代码段将在页请求出错时被调用，基本上是释放请求的页。

第 1629~1632 行：启用中断，这样就允许中断传出了。

4.5.4 Slab 的销毁：退还内存与 `kmem_cache_destroy()`

缓存与 slab 都可被销毁。缓存可被缩减或者销毁以返还其占用的内存给空闲内存池。内核在系统内存过低时会调用析构函数。在任意一种情况下，slab 也可被销毁且将与之对应的页面返还给伙伴系统，以便重复利用。函数 `kmem_cache_destroy()` 用来销毁缓存。下面我们会详细分析该函数。函数 `kmem_cache_reap()` (`mm/slab.c`) 和 `kmem_cache_shrink` (`mm/slab.c`) 分别回收和缩减缓存。其中与伙伴系统的交互是通过函数 `kmem_freepages()` (`mm/slab.c`) 完成的。

`kmem_cache_destroy()`

如果要删除缓存，有几个实例可以使用。动态加载模块（假定载入和卸载过程中没有固定的永久内存预留）创建的缓存就必须在卸载时被销毁，从而可以释放内存，确保在下次模块载入时不出现重复缓存。因此，通常以这种方式销毁专用缓存。

销毁缓存的步骤与创建缓存的步骤相逆。在销毁缓存时不用关心对齐问题，只需要删除缓存描述符并释放内存即可。销毁缓存步骤可概括如下：

- (1) 从缓存链表中删除缓存；
- (2) 删除 slab 描述符；
- (3) 删除缓存描述符。

```
-----
mm/slab.c
1421 int kmem_cache_destroy (kmem_cache_t * cachep)
1422 {
1423     int i;
1424
1425     if (!cachep || in_interrupt())
1426         BUG();
1427
1428     /* Find the cache in the chain of caches. */
1429     down(&cache_chain_sem);
1430     /*
1431      * the chain is never empty, cache_cache is never destroyed
1432      */
1433     list_del(&cachep->next);
1434     up(&cache_chain_sem);
1435
1436     if (__cache_shrink(cachep)) {
1437         slab_error(cachep, "Can't free all objects");
1438         down(&cache_chain_sem);
1439         list_add(&cachep->next, &cache_chain);
1440         up(&cache_chain_sem);
1441         return 1;
1442     }
1443
```

```

...
1450  kmem_cache_free(&cache_cache, cachep);
1451
1452  return 0;
1453  }

```

该函数的参数 `cache` 是一个指针，指向待销毁缓存的描述符。

第 1425~1426 行：该代码段完成完整性检查，其中包括确认程序当前不在中断上下文中，并且缓存描述符不为空。

第 1429~1434 行：获得 `cache_chain` 信号量，从缓存链中删除指定缓存，释放 `cache_chain` 信号量。

第 1436~1442 行：该代码段负责释放未使用的 slab。若函数 `_cache_shrink()` 返回真，则表明缓存中仍然存在有 slabs，因此不可销毁缓存。这样要求我们按相反的顺序执行前面的步骤，重新将缓存描述符插入 `cache_chain` 链表中，当然同样需要先获取 `cache_chain` 信号量，操作完成后再释放它。

第 1450 行：最后，释放缓存描述符，结束操作。

4.6 内存请求路径

到目前为止，虽然我们已经讨论了 slab 分配器，但还未与任何实际的内存请求结合起来。除了缓存初始化函数外，我们还没有说明如何把这些函数配合在一起调用。因此，从现在开始我们来跟踪有关的内存请求的控制流程。当内核必须获得字节大小分组的内存块时，就需要使用函数 `kmalloc()`——它实际上会调用到函数 `kmem_getpages` 完成实际分配。调用路径如下所示：

```
kmalloc()->__cache_alloc()->kmem_cache_grow()->kmem_getpages()
```

4.6.1 kmalloc()

`kmalloc()` 函数在内核中分配内存对象：

```

-----
mm/slab.c
2098  void * __kmalloc (size_t size, int flags)
2099  {
2100      struct cache_sizes *csizes = malloc_sizes;
2101
2102      for (; csizes->cs_size; csizes++) {
2103          if (size > csizes->cs_size)
2104              continue;
2105      }
2106      ...
2112      return __cache_alloc(flags & GFP_DMA ?
2113          csizes->cs_dmacachep : csizes->cs_cachep, flags);
2114  }
2115      return NULL;
2116  }
-----

```

1. size

size 是请求的字节数。

2. flags

flags 指定内存请求的类型。这些标志将被传递给伙伴系统而不会影响 `kmalloc()` 的行为。

表 4-6 对这些标志进行了描述，有关它们的细节请参见 4.3.3 节。

表4-6 `vm_area_struct->vm_flags`的值

标 志	描 述
VM_READ	表示该区域中的页可读
VM_WRITE	表示该区域中的页可写
VM_EXEC	表示该区域中的页可执行
VM_SHARED	表示与其他进程共享该区域中的页
VM_GROWSDOWN	表示线性地址向低端内存扩展
VM_GROWSUP	表示线性地址向高端内存扩展
VM_DENYWRITE	表示该区域的页不可写
VM_EXECUTABLE	表示该区域内包含可执行代码
VM_LOCKED	表示页面被锁定
VM_DONTCOPY	表示页面不可被克隆
VM_DNTEXPAND	表示不可扩展该虚拟内存区

第 2102~2104 行：在查找缓存的过程中，找到所包含的对象大小大于所需大小的第一个缓存。

第 2112~2113 行：从 flags 参数指定的内存管理区中分配对象。

4.6.2 `kmem_cache_alloc()`

该函数是对函数 `__cache_alloc()` 的包装。它实际不实现任何额外的功能，其参数按如下方式传递给函数：

```
-----
mm/slab.c
2070 void * kmem_cache_alloc (kmem_cache_t *cachep, int flags)
2071 {
2072     return __cache_alloc(cachep, flags);
2073 }
-----
```

1. cachep

cachep 参数是我们的待分配对象的缓存描述符。

2. flags

flags 表示内存请求方式，它是按照函数 `kmalloc()` 的要求直接传递过来的。

内核提供的函数 `kfree()` 接口用于释放由 `kmalloc` 分配的字节大小的内存块，它接收的参数为指向由 `kmalloc()` 执行完毕所返回的内存的指针。图 4-9 描述了由 `kfree` 到 `kmem_freepages` 的调用过程。

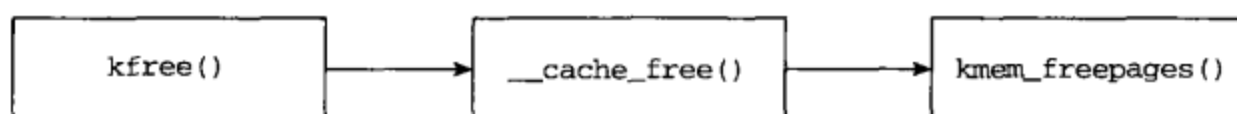


图 4-9 kfree()的调用图

4.7 Linux 进程的内存结构

到现在为止，我们已经讨论了内核如何管理自己的内存。现在要将注意力转向用户空间程序，观察内核如何管理它们使用的内存。在虚拟内存的帮助下，用户空间进程可有效地操作内存，就好像它可以访问系统中所有内存一样。事实上，内核会负责管理载入什么，如何载入以及何时进行这些操作。本节所讨论的内容就是内核如何管理内存，以及如何能对用户空间程序完全透明。

用户进程创建后便要分配一个虚拟地址空间。正如我们以前所讲的，进程的虚拟地址空间是一段未分段的、进程可访问的线性地址范围。这段地址的大小取决于系统体系结构中寄存器的大小。大多数系统有 32 位的地址空间，有些系统（如 A G5）的处理器具有 64 位的地址空间。

进程被创建后，其地址范围可以通过增加或删除线性地址间隔（address interval）得以扩大或缩减。地址间隔（一段地址区间）是一种内存单元，它也被称为内存范围（memory region）或内存区（memory area）。把进程地址空间划分成不同类型的区域是有用的，不同的内存区具有不同的保护方案和特点。进程内存区的保护方案与进程上下文相关，比如程序代码段的某些部分标记为只读（text），而其他部分则标记为可写（变量）或可执行（指令）。同样，一个特殊进程可能仅能访问属于它的内存区。

在内核中，进程地址空间以及与之相关的所有信息都保存在 `mm_struct` 描述符中。回想一下第 3 章，该结构出现在进程控制结构 `task_struct` 中。一个内存区由 `vm_area_struct` 描述符表示。每个内存区描述符都描述它所表示的一段连续地址区间。本章中，我们将称地址区间描述符为内存区间描述符或 `vma_area_struct`。接下来看看 `mm_struct` 和 `vm_area_struct` 结构。图 4-10 描述了这些数据结构之间的关系。

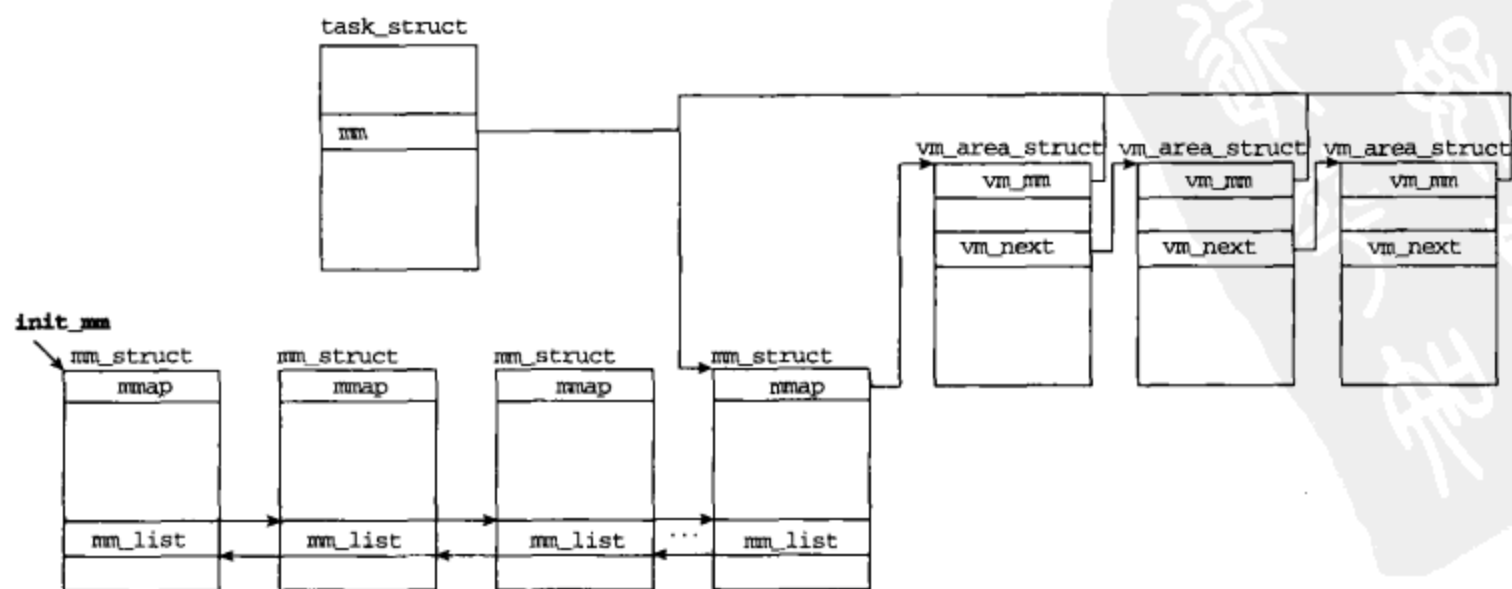


图 4-10 进程相关的内存结构体

4.7.1 mm_struct

每一个任务都有一个 mm_struct (include/linux/sched.h) 结构, 内核用它来表示它自己的内存地址范围。所有 mm_struct 描述符统一存放在一个双向链表中。链表头是对应于进程 0 的 mm_struct, 进程 0 是空闲进程, 可通过全局变量 init_mm 来访问该描述符。

```
-----
include/linux/sched.h
185 struct mm_struct {
186     struct vm_area_struct * mmap;
187     struct rb_root mm_rb;
188     struct vm_area_struct * mmap_cache;
189     unsigned long free_area_cache;
190     pgd_t * pgd;
191     atomic_t mm_users;
192     atomic_t mm_count;
193     int map_count;
194     struct rw_semaphore mmap_sem;
195     spinlock_t page_table_lock
196
197     struct list_head mmlist;
198     ...
202     unsigned long start_code, end_code, start_data, end_data;
203     unsigned long start_brk, brk, start_stack;
204     unsigned long arg_start, arg_end, env_start, env_end;
205     unsigned long rss, total_vm, locked_vm;
206     unsigned long def_flags;
207     cpumask_t cpu_vm_mask;
208     unsigned long swap_address;
209     ...
228 };
-----
```

1. mmap

进程所对应的内存区描述符 (下一节给出定义) 被分配给链接于一个链表中的进程。通过 mm_struct 的 mmap 字段访问该链表, 而 vma_area_struct 中的 vm_next 字段将各个内存区描述符结点链接起来。

2. mm_rb

虽然用简单链表链接内存描述符是检索指定进程对应的内存区描述符最方便、简单的方法。但是这样搜索一个指定的内存描述符, 简单链表是无法提供高性能检索的。所以进程地址空间对应的内存描述符是以红黑树的形式存放的, mm_rb 字段便指向树根节点。利用红黑数存放结点确保了内核能以最快速度访问指定的内存区描述符。

3. mmap_cache

mmap_cache 是指向进程最后一个内存区描述符的指针。由于内存地址访问的局部性原理 (principle of locality), 临近于最后访问地址的内存位置最有可能接下来被很快访问到。于是, 很有可能当前检查的地址和最后一次检查地址是在同一个内存区。要知道当前地址和最后一次访问地址处于同一个内存区的可能性约为 35%。

4. pgd

Pgd 是一个指针, 指向存放该内存区中页的页全局目录。在空进程(进程 0) 对应的 mm_struct 中, 该字段指向 swapper_pg_dir 目录。请参见 4.9 节了解该字段的更多信息。

5. mm_users

mm_users 字段存放访问该内存区的进程数量。轻量进程或者线程共享同一地址区间和内存区, 所以线程对应的 mm_struct 其 mm_users 字段值通常大于 1。通过以下原子函数对该字段进行操作: atomic_set()、atomic_dec_and_lock()、atomic_read() 和 atomic_inc()。

6. mm_count

mm_count 字段是对 mm_struct 结构的使用统计。当确定 mm_struct 结构是否可以被释放时, 就需要检查该字段。如果其值为 0, 则说明没有进程正在使用它, 因此可以将其回收。

7. map_count

map_count 字段存放在进程地址空间中的内存区数量, 或者说 vma_area_struct 描述符的数量。每当一个新内存区加入到进程地址空间时, 随着 vma_area_struct 结构插入 mmap 链表和 mm_rb 树中, 该字段的值加 1。

8. mm_list

类型为 list_head 的 mm_list 字段存放内存描述符链表中相邻的 mm_structs 结构的地址。如前面所说链表头由全局变量 init_mm 指向, 它是进程 0 的内存描述符。当操作该链表时, 需要 mm_list_lock 锁, 保护其不被并发访问。

接下来讨论的 11 个字段用于处理进程需要分配给它们的各种内存区。为了避免偏离讨论进程内存相关结构这一主题, 我们只对这些字段作粗略的描述。

9. start_code 和 end_code

start_code 和 end_code 字段存放进程在内存中代码段 (也就是可执行的正文段) 的开始地址和结束地址。

10. start_data 和 end_data

start_data 和 end_data 字段存放初始化数据 (可执行文件中 .data 部分) 的起始地址和结束地址。

11. start_brk 和 brk

start_brk 和 brk 字段存放进程堆的开始和结束地址。

12. start_stack

start_stack 存放进程栈的开始地址。

13. arg_start 和 arg_end

arg_start 和 arg_end 存放传递给进程的参数的起始和结束地址。

14. env_start 和 env_end

env_start 和 env_end 字段存放环境段的开始和结束地址。

本章主要讨论了 mm_struct 中的各个字段的含义。接下来我们来看看内存区描述符 vm_area_struct 中的各个字段。

4.7.2 vm_area_struct

vm_area_struct 结构定义了虚拟内存区。进程具有各种不同的内存区，但是每个内存区都只对应一个 vm_area_struct 结构。

```
-----
include/linux/mm.h
51 struct vm_area_struct {
52     struct mm_struct * vm_mm;
53     unsigned long vm_start;
54     unsigned long vm_end;
...
57     struct vm_area_struct *vm_next;
...
60     unsigned long vm_flags;
61
62     struct rb_node vm_rb;
...
72     struct vm_operations_struct * vm_ops;
...
};
-----
```

1. vm_mm

所有的内存区都属于进程的地址空间，而地址空间由 mm_struct 表示。结构 vm_mm 便指向该内存区所在的地址空间的 mm_struct 结构。

2. vm_start和vm_end

内存区都和一个地址区间关联。在 vm_area_struct 结构中每个地址区间都有自己的起始和结束位置。考虑到性能问题，内存区的开始地址必须是页面大小的整数倍，而且内核确保用指定内存区中的数据填充页面时也是按照页面大小的整数倍进行。

3. vm_next

vm_next 指向链表中下一个 vm_area_struct 结构，链表包含进程地址空间所有的内存区。链表头是由进程地址空间对应的 mm_struct 结构中 mmap 字段所指向的。

4. vm_flags

内存区间具有自己相关的特性，这些特性都存放在 vm_flags 字段中，并适用于内存区中的页。表4-6描述了这些标志的可能取值。

5. vm_rb

vm_rb 存放该内存区对应的红黑树节点。

6. vm_ops

vm_ops 字段是由函数指针组成的结构体，这些函数用于操作特定的 vm_area_struct，包括打开内存区、关闭内存区、反映射内存区等，另外还包含一个函数指针，缺页异常发生时调用该函数。

4.8 进程映像的分布及线性地址空间

当用户空间程序被载入内存后，它就拥有自己的线性地址空间，该空间被划分成各种内存区或叫做段 (segment)。按照进程执行功能上的差异，线性空间被划分成一个个段。这些功能上独立的段被映射到进程地址空间中。与进程执行相关的主要段有 6 个。

- ❑ **text**: 该段也被称作代码段，它存放程序的可执行指令。因此它具有 `execute` 和 `read` 属性。如果同一程序被实例化为多个进程，则同一指令载入两次必然造成内存浪费，因此 Linux 允许多进程共享 **text** 段。`mm_struct` 中的 `start_code` 和 `end_code` 字段保存了 `text` 段的起始位置和结束位置。
- ❑ **Data**: 该段存放所有已初始化的数据。初始化数据包含静态分配的数据和初始化的全局数据。下面的代码段给出了初始化数据的示例。

```
-----  
example1.c  
int gvar = 10;  
  
int main(){  
...  
}
```

- ❑ **gvar**: 全局变量，它被初始化后存放在数据段中。该段具有可读、可写属性，但是不可被运行在同一程序中的其他进程共享。`mm_struct` 结构的 `start_data` 和 `end_data` 字段存放数据段的起始地址和结束地址。
- ❑ **BSS**: 该段存放未初始化数据。这些数据包括程序执行时被系统初始化成 0 的全局变量。该段的另一个名字叫做初始化为 0 的数据段。下面的代码段给出了未初始化的数据示例：

```
-----  
example2.c  
int gvar1[10];  
long gvar2;  
  
int main() {  
...  
}
```

该段内的对象仅仅有名称和大小属性。

- ❑ **堆 (heap)**, 该段用于扩展进程的线性地址空间。当程序调用函数 `malloc()` 获取动态内存时，新获取的动态内存便被置于堆中。`mm_struct` 结构的 `start_brk` 和 `brk` 字段保存堆的起始地址和结束地址。当 `malloc` 被调用获得动态内存时，系统调用 `sys_brk()` 将 `brk` 指针移动到新位置，从而扩展了堆空间。
- ❑ **栈 (stack)**, 该段包含所有已分配内存的局部变量。当函数被调用时，函数的局部变量被压入栈。当函数结束时，与函数相关的变量被弹出栈。其他信息，包括返回地址和参数同样也被压入栈。`mm_struct` 结构中的 `start_stack` 字段标记了进程栈的起始位置。

虽然进程执行与上述 6 个主要段相关，但它们只映射到地址空间中的 3 个内存区。这些内存区分别叫做 text data 和 stack。data 段包括可执行程序的初始化 data 段、bss 和堆，text 段包含可执行程序的 text 段。图 4-11 描述了线性地址空间的结构以及 mm_struct 如何跟踪记录这些段。

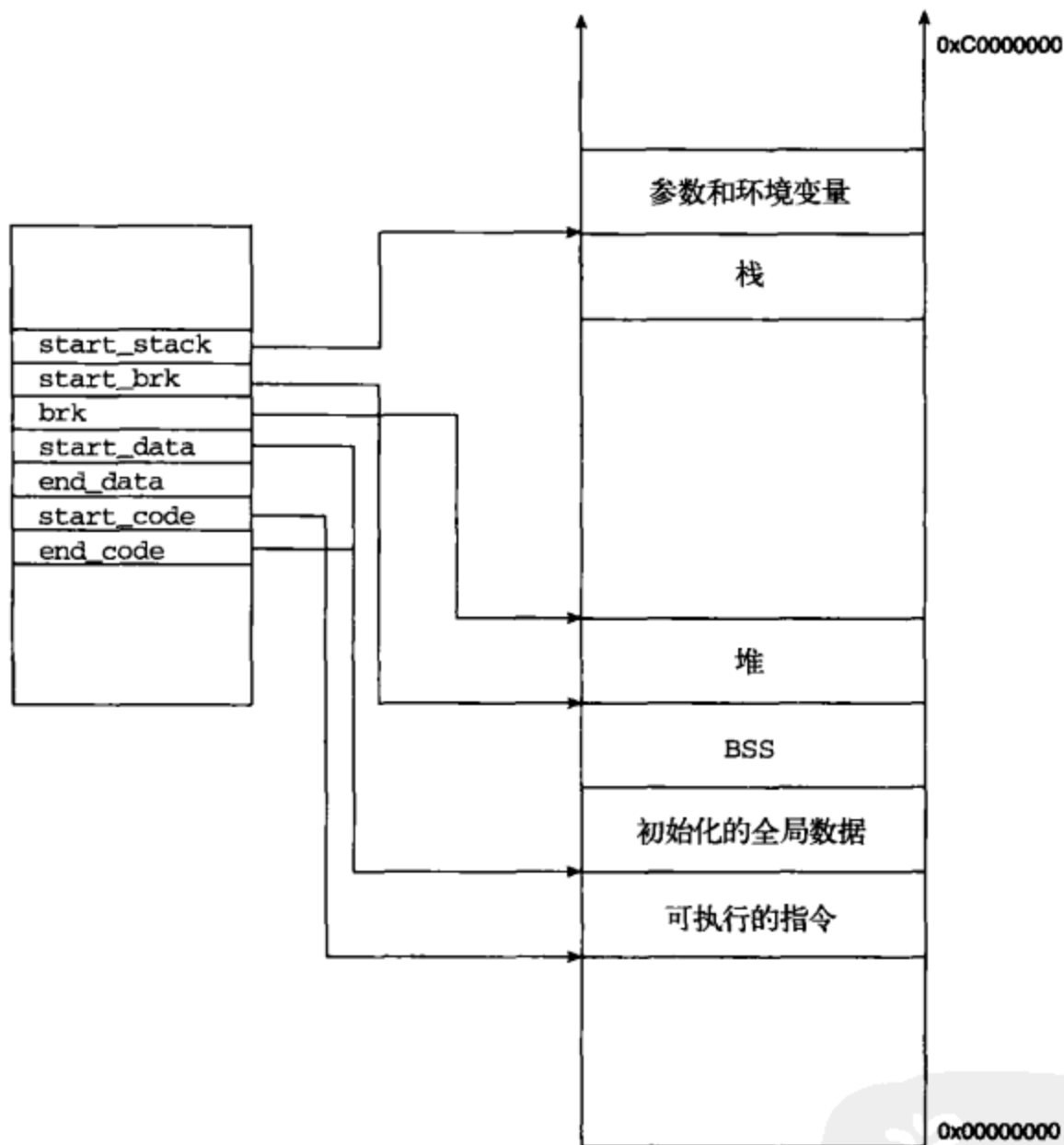


图 4-11 进程地址空间

各种内存区都被映射到 /proc 文件系统中，可通过 /proc/<pid>/maps 的输出访问进程的内存映射。我们通过一个例子来看看进程地址空间中的所有内存区。example3.c 中的代码为我们演示了被映射的程序。

```
-----
example3.c
#include <stdio.h>
int main(){
    while(1);
    return(0);
}
-----
```

图 4-12 描述了示例程序执行后输出的 `/proc/<pid>/maps` 中的内容。

```

08048000-08049000 r-xp 00000000 03:05 1324039 /home/1kp/example3
08049000-0804a000 rw-p 00000000 03:05 1324039 /home/1kp/example3
40000000-40015000 r-xp 00000000 03:05 767058 /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 03:05 767058 /lib/ld-2.3.2.so
40016000-40017000 rw-p 00000000 00:00 0
42000000-4212e000 r-xp 00000000 03:05 1011853 /lib/tls/libc-2.3.2.so
4212e000-42131000 rw-p 0012e000 03:05 1011853 /lib/tls/libc-2.3.2.so
42131000-42133000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp fffff000 00:00 0

```

图 4-12 `cat/proc/<pid>/maps`

最左边的列显示的是内存段的范围，也就是段的起始地址和结束地址。下一列显示的是段的访问权限。这些权限标识类似于文件访问权限：`r` 表示可读、`w` 表示可写、`x` 表示可执行，最后一个权限可能是 `p` 表示私有段，或者 `s` 表示共享段（一个私有段并非必然是不可共享的）。`p` 表示仅当前不可共享。下一列表示段的偏移量。左起第 4 列为由冒号分割的两个数字，它们分别代表与段相关的文件在文件系统中的主设备号从设备号（有些段不存在对应文件，所以该列值为 `00:00`）。第 5 列表示文件对应的索引节点，第 6 列和最右边的列存放文件名。如果段没有对应的文件名，那么该列为空，而且索引节点列为 0。

在我们的例子中，第一行是实例程序中代码段的描述。可以看到它的访问权限为可执行，下一行描述了实例程序的数据段，注意它的访问权限为可写。

我们实例程序被动态链接，这意味着它用到的库函数在运行时才被加载。这些库函数也需要被映射到进程地址空间才能访问。下面 6 行便描述了动态链接库的段。开始 3 行依次为 `ld` 库的代码段、数据段和 `bss` 段。在下面三行则依次是 `libc` 的代码段、数据段和 `bss` 段。

最后一行，其权限表示它可读、可写、可执行，它代表的是进程栈空间，最高可扩展到 `0xC0000000`，`0xC0000000` 是用户空间进程可访问的最高内存地址。

4.9 页表

程序内存的有效管理是通过虚拟地址完成的。但现在面临的问题是，当把指令提交给处理器时，处理器却对虚拟地址无能为力。处理器只能操作物理地址。虚拟地址和对应的物理地址之间的管理便要靠内核（借助于硬件）中的页表来维护。

页表对内存中的页面走向进行记录。在内核运行的整个生命期中，页表都存放在内存中。Linux 有称为 3 级页表的分页机制。3 级页表确保 64 位体系结构有足够的空间维护它们的虚拟地址到物理地址的转换。从名字我们可以看出，3 级页表结构有三种不同的页表：顶层目录称作 PGD (Page Global Directory)，它由数据类型 `pdg_t` 表示；第 2 级页表称为 PMD (Page Middle Directory)，它由数据类型 `pmd_t` 表示；最后一级页表称为 PTE (Page Table)，它由数据类型 `pte_t` 表示。图 4-13 描述页表结构。

PGD 存放的项指向 PMD，PMD 存放的项指向 PTE，而 PTE 存放的项指向特定页面。每个进程都有自己的页表。`mm_struct->pgd` 字段指向进程的 PGD。32 位或 64 位的虚拟地址被分成各

种大小（取决于体系结构）的偏移字段。这些字段对应于 PGD、PMD、PTE 和页面本身的偏移。

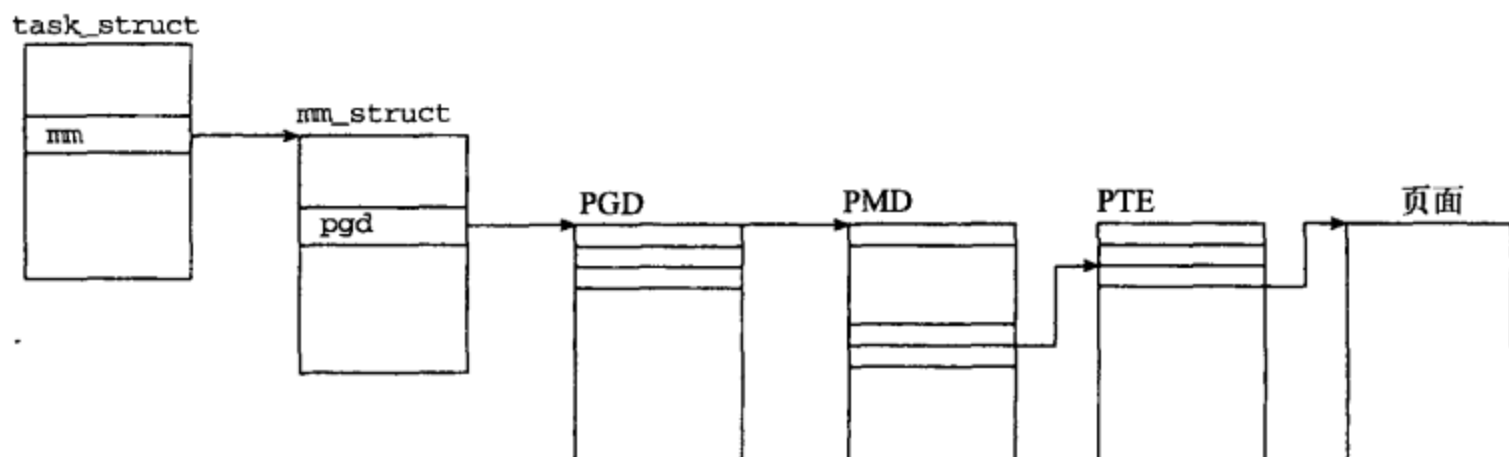


图 4-13 Linux 的页表结构

4.10 缺页

在进程运行期间，很有可能发生所访问的地址在进程地址空间，但并不在内存的情况；也有可能要访问的页在内存，但访问行为与页设置不符（比如对只读区域进行写操作）。当上述情况发生时，系统就产生缺页（page fault）。缺页是一种异常处理程序，它管理在程序页访问期间所发生的错误。当硬件发起（内核捕获的）缺页异常时，页将被从存储设备取回，内核就分配缺失的页。

每种体系结构都有一个与体系结构相关的函数处理缺页。x86 和 PPC 都调用函数 `do_page_fault()`。x86 的缺页处理函数 `do_page_fault(*regs, error_code)` 位于文件 `/arch/i386/mm/fault.c` 中；PowerPC 缺页处理函数 `do_page_fault(*regs, address, error_code)` 位于 `/arch/ppc/mm/fault.c` 文件中。这两个函数的功能非常相似，所以我们只需要讨论 x86 的 `do_page_fault()` 函数，就可了解 PowerPC 版本函数的功能。

两种体系结构处理缺页的不同主要在于调用函数 `do_page_fault()` 前，错误信息如何获取和保存。我们首先解释 x86 缺页处理的特点，接着将解释 `do_page_fault()` 函数。结束时我们将解释它与 PowerPC 的区别。

4.10.1 x86 缺页异常

x86 的缺页处理函数 `do_page_fault()` 在硬件中断 14 发生时被调用。在处理器发现下列条件为真时该中断发生。

- (1) 允许请页，该地址的页目录或页表项的存在位为 0。
 - (2) 允许请页，当前特权级（特权级别数字低者特权高）低于访问请求页所需的特权。
- 一旦缺页中断发生，处理器保存两个信息。

- (1) 错误的信息被压栈，错误信息存放在字的低 4 位（`do_page_fault()` 函数未使用第 3 位）。

参见表 4-7 了解各位值的对应含义。

表4-7 缺页error_code

	位 2	位 1	位 0
Value = 0	内核	读	页不存在
Value = 1	用户	写	保护错误

(2) 引起异常的 32 位线性地址，该地址存放在寄存器 cr2 中。

函数 do_page_fault() 的参数 regs 是一个结构体，其中包含的是系统寄存器，error_code 参数使用其中低 3 位描述缺页的原因。

4.10.2 缺页处理程序

在两种体系结构中，do_page_fault() 函数都使用给定信息，且执行几种特定操作之一。这些代码段按照相当复杂的流程检查缺页，并按下列几种方式之一完成操作。

- 由 handle_mm_fault() 函数找到缺页发生的偏移地址。
- 著名的 oops 转储（在标记 no_context: 处），Powerpc 的 bad_page_fault()。
- 产生一个段错误，（在标记 bad_area: 处），PowerPc 调用 bad_page_fault() 处理。
- 返回错误给调用者（修正）。

```
-----
arch/i386/mm/fault.c
212  asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
213  {
214      struct task_struct *tsk;
215      struct mm_struct *mm;
216      struct vm_area_struct * vma;
217      unsigned long address;
218      unsigned long page;
219      int write;
220      siginfo_t info;
221
222      /* get the address */
223      __asm__("movl %%cr2,%0":"=r" (address));
224      ...
232      tsk = current;
233
234      info.si_code = SEGV_MAPERR;
-----
```

第 223 行：发生缺页的地址被存放在 cr2 控制寄存器中，该线性地址被读入，并设置本地变量 address 来保存该值。

第 232 行：把 task_struct 类型指针 tsk 设置为指向当前任务的 task_struct 结构。

现在，我们可以找到缺页产生的地址，图 4-14 描述下列代码段的执行流程。

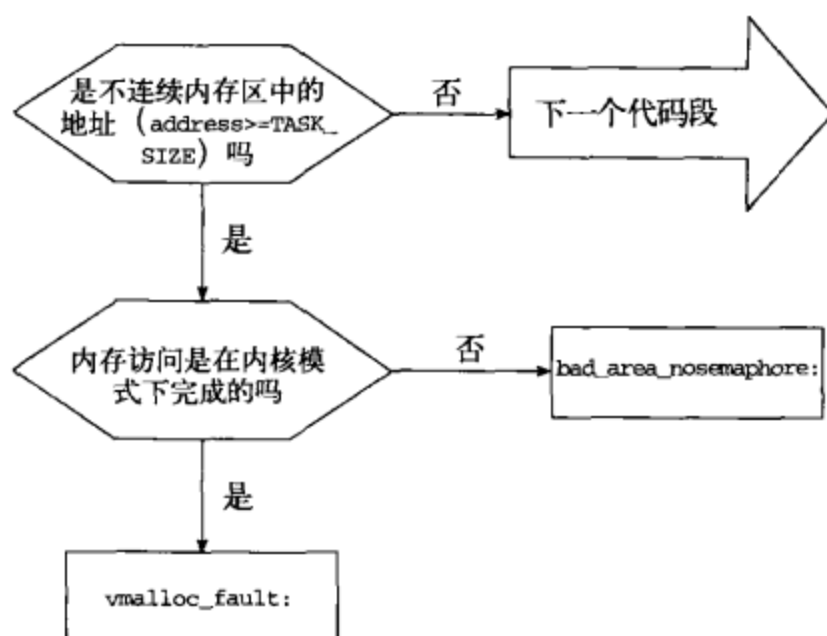


图 4-14 缺页 I

```

-----
arch/i386/mm/fault.c
246  if (unlikely(address >= TASK_SIZE)) {
247      if (!(error_code & 5))
248          goto vmalloc_fault;
...
253  goto bad_area_nosemaphore;
254  }
...
257  mm = tsk->mm
...
-----

```

第 246~248 行：该代码段检查发生缺页的地址是否位于内核模块的空间（也就是说处于非连续内存区中）。非连续内存区地址的线性地址 $\geq \text{TASK_SIZE}$ 。如果确实发生在非连续区，则继续检查 `error_code` 的 0 位和 2 位是否为空。回想表 4-7 可知，该错误是由于访问不存在的内核页面造成的。如果真是这样，则说明缺页发生在内核态，于是标记 `vmalloc_fault:` 处的代码将被调用。

第 253 行：如果我们执行到这一行，则意味着虽然访问发生在非连续内存区，但它仍然发生在用户态，或触发了保护错误，或两者都有。在这种情况下，跳到标记 `bad_area_nosemaphore:` 处执行。

第 257 行：该行代码设置局部变量 `mm` 指向当前任务的内存描述符，如果当前任务是内核线程，那么该值为 `NULL`，这点在下面的代码中变得很有意义。

此刻，我们已经确定了缺页不是发生在非连续内存区，图 4-15 描述下面代码的执行流程。

```

-----
arch/i386/mm/fault.c
...
262  if (in_atomic() || !mm)
263      goto bad_area_nosemaphore;

```

```

264
265  down_read(&mm->mmap_sem);
266
267  vma = find_vma(mm, address);
268  if (!vma)
269    goto bad_area;
270  if (vma->vm_start <= address)
271    goto good_area;
272  if (!(vma->vm_flags & VM_GROWSDOWN))
273    goto bad_area;
274  if (error_code & 4) {
...
281    if (address + 32 < regs->esp)
282      goto bad_area;
283  }
284  if (expand_stack(vma, address))
285    goto bad_area;
...

```

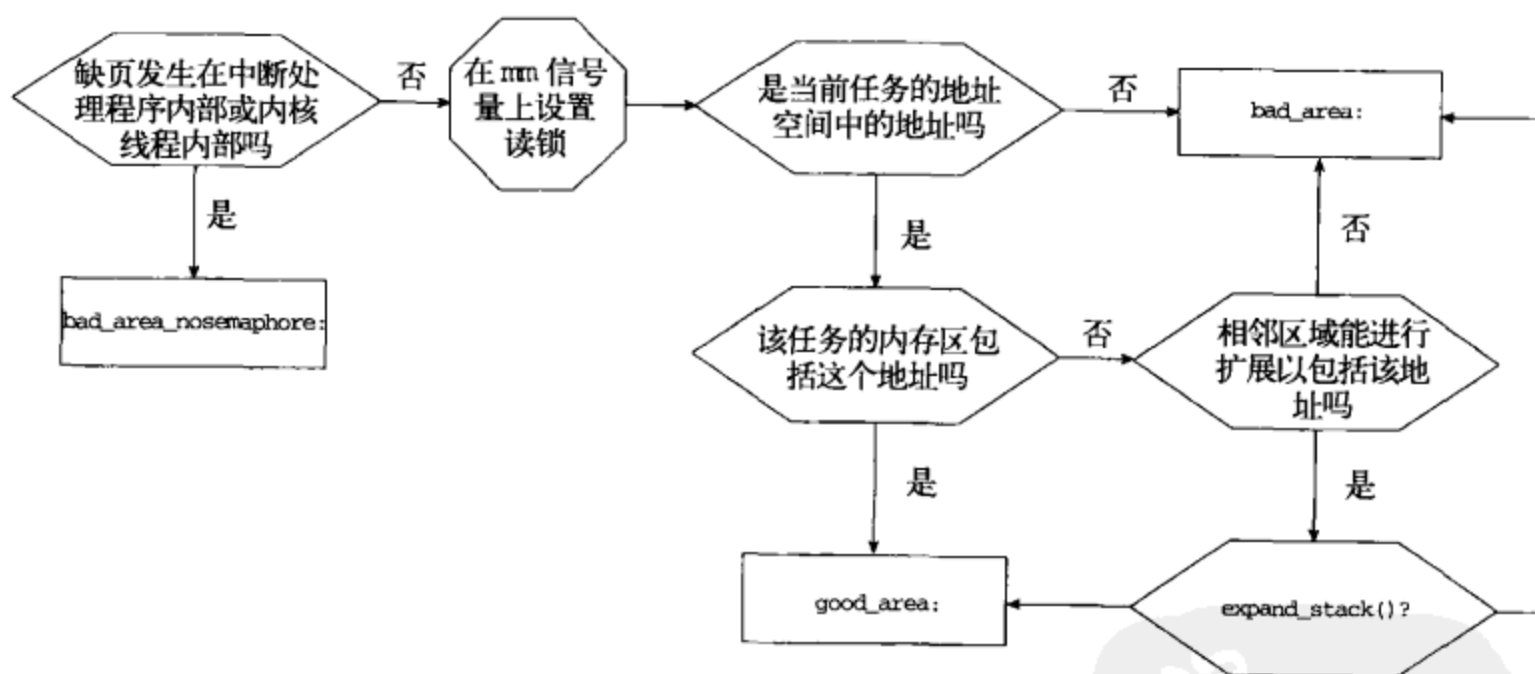


图 4-15 缺页 II

第 262~263 行：该代码段检查缺页是否发生于中断处理程序或内核空间中，如果是，那么我们转到标记 `bad_area_semaphore:` 处执行。

第 265 行：这里我们将准备搜索当前进程的内存区，所以我们在内存描述符的信号量上设置读锁。

第 267~269 行：在这里，假定我们知道缺页产生的地址不是在内核线程或中断处理程序中，于是就搜索进程地址空间查看这个地址是否处于进程的某个内存区，如果不在，则跳到标记 `bad_area:` 处执行。

第 270~271 行：如果在进程地址空间中找到了一个可用的内存区，则跳到标记 `good_area:` 处执行。

第 272~273 行：如果找到的内存区不可用，那么检查最临近的区间是否可扩展，以满足页请

求, 如果不可, 则跳到标记 `bad_area:` 处执行。

第 274~284 行: 否则, 出错地址可能由栈操作引起, 如果扩展栈没有什么帮助, 则跳到标记 `bad_area:` 处执行。

接下来, 继续解释跳转标记点处代码的意义。我们从标记 `vmalloc_fault` 处开始解释。如图 4-16 所示。

```
-----
arch/i386/mm/fault.c
473 vmalloc_fault:
{

    int index = pgd_index(address);
    pgd_t *pgd, *pgd_k;
    pmd_t *pmd, *pmd_k;
    pte_t *pte_k;

    asm("movl %%cr3,%0":"=r" (pgd));
    pgd = index + (pgd_t *)__va(pgd);
    pgd_k = init_mm.pgd + index;

491    if (!pgd_present(*pgd_k))
        goto no_context;

    pmd = pmd_offset(pgd, address);
    pmd_k = pmd_offset(pgd_k, address);
    if (!pmd_present(*pmd_k))
        goto no_context;
    set_pmd(pmd, *pmd_k);

    pte_k = pte_offset_kernel(pmd_k, address);
506    if (!pte_present(*pte_k))
507        goto no_context;
508    return;
509 }
```

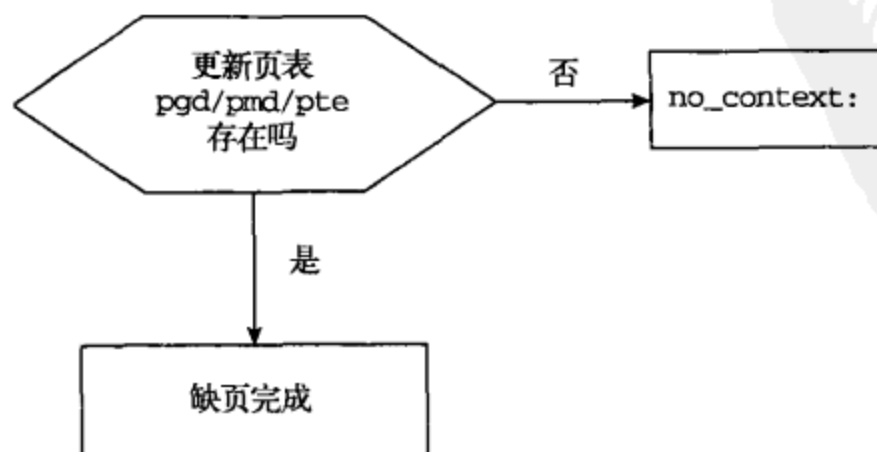


图 4-16 标记 `vmalloc_fault`

第 473~509 行: 当前进程的页全局目录被获取 (通过 `cr3` 寄存器) 并且存放在变量 `pgd` 中,

内核页全局目录被取得并放入变量 `pgd_k` (按同样的方式处理 `pmd` 和 `pte` 变量) 中。如果出错地址不在内核分页系统中, 则代码跳到标记 `no_context:` 处, 否则当前进程使用内核的 `pgd`。

现在, 我们看看标记 `good_area:`。此刻我们知道保存出错地址的内存区在进程的地址空间中, 接下来需要确认地址的访问权限是否正确。图 4-17 给出了流程图。

```
-----
arch/i386/mm/fault.c
290 good_area:
291     info.si_code = SEGV_ACCERR;
292     write = 0;
293     switch (error_code & 3) {
294         default: /* 3: write, present */
295     ...
        /* fall through */
300     case 2: /* write, not present */
301         if (!(vma->vm_flags & VM_WRITE))
302             goto bad_area;
303         write++;
304         break;
305     case 1: /* read, present */
306         goto bad_area;
307     case 0: /* read, not present */
308         if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
309             goto bad_area;
310     }
```

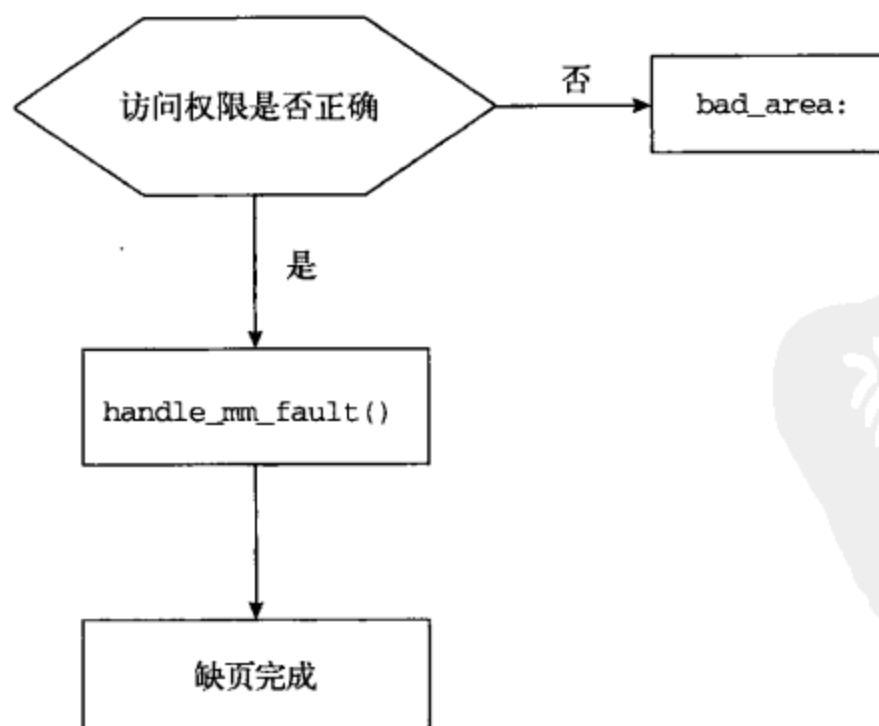


图 4-17 标记 `good_area`

第 294~304 行: 如果缺页是由一个内存读操作引起的 (回想如果是这种情况, 则 `error code` 最左边的位被设置为 1)。我们检查内存区是否是可写的, 如果是不可写的, 那么说明访问权限不匹配, 跳到标记 `bad_area:` 处执行; 如果是可写的, 则依次判断各种 `case` 情况, 并最终进入

函数 `handle_mm_fault()` 执行, 同时将局部变量 `write` 置为 1。

第 305~309 行: 如果缺页是由读或执行访问触发的, 而且页存在。那么我们跳转到标记 `bad_area:` 处执行, 因为显而易见访问权限冲突。如果页不存在, 则检查内存区是否具有可读或可执行属性。如果没有, 依然跳转到标记 `bad_area:` 处, 因为即使要取回页, 访问权限保护依然不允许访问; 如果有, 依次判断各种 `case` 情况, 最终进入函数 `handle_mm_fault()` 执行, 同时局部变量 `write` 被置 0。

下面标记处的代码是在权限检查通过后被执行的。它被标记为 `survive:`

```
-----
arch/i386/mm/fault.c
survive:
318  switch (handle_mm_fault(mm, vma, address, write)) {
    case VM_FAULT_MINOR:
        tsk->min_flt++;
        break;
    case VM_FAULT_MAJOR:
        tsk->maj_flt++;
        break;
    case VM_FAULT_SIGBUS:
        goto do_sigbus;
    case VM_FAULT_OOM:
        goto out_of_memory;
329  default:
        BUG();
    }
-----
```

第 318~329 行: 函数 `handle_mm_fault()` 被调用时, 所提供的参数有: 当前内存描述符 (`mm`)、出错地址所在的内存区描述符、出错地址, 以及访问是否可读、可执行或可写。switch 帮助我们捕捉错误, 确保正确地退出程序执行。

下面的代码片描述了标记 `bad_area` 和 `bad_area_no_semaphore`。当跳到该处执行时, 我们知道下列条件至少其中之一发生。

- (1) 产生缺页的地址不在进程地址空间, 因为已搜索了内存区, 并未发现有匹配的。
 - (2) 产生缺页的地址不在进程地址空间, 而且该地址所在的内存区不能扩展来存放它。
 - (3) 产生缺页的地址在进程地址空间, 但内存区的访问权限与希望执行的操作不匹配。
- 现在需要确定访问是否发生在内核态, 下面的代码和图 4-18 描述这些标记的处理流程。

```
-----
arch/i386/mm/fault.c
348  bad_area:
349  up_read(&mm->mmap_sem);
350
351  bad_area_nosemaphore:
352  /* User mode accesses just cause a SIGSEGV */
353  if (error_code & 4) {
354  if (is_prefetch(regs, address))
355  return;
356
```

```

357     tsk->thread.cr2 = address;
358     tsk->thread.error_code = error_code;
359     tsk->thread.trap_no = 14;
360     info.si_signo = SIGSEGV;
361     info.si_errno = 0;
362     /* info.si_code has been set above */
363     info.si_addr = (void *)address;
364     force_sig_info(SIGSEGV, &info, tsk);
365     return;
366 }

```

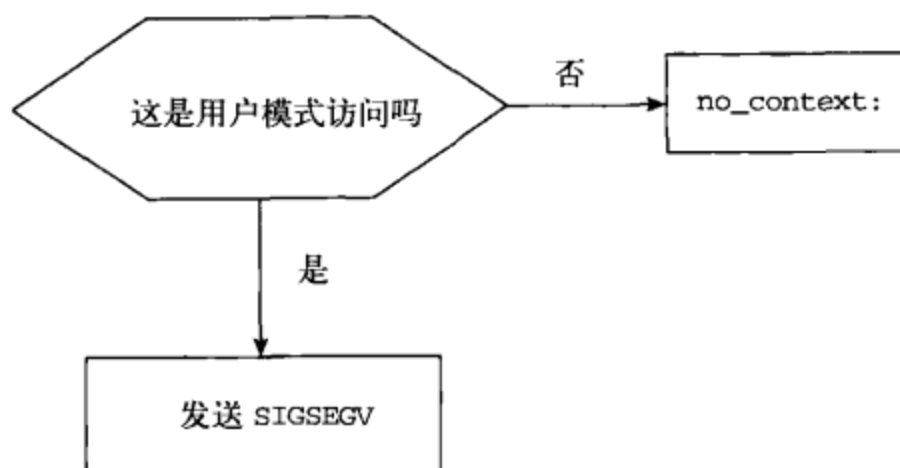


图 4-18 标记 bad_area

第 348 行：函数 `up_read()` 释放进程内存描述符信号量上的读锁。注意，当我们在内存描述符的信号量加上读锁，以遍历内存区，从而查看地址是否处于进程地址空间中之后，只跳到标记 `bad_area:` 处执行；否则跳入到标记 `bad_area_nosemaphore:` 处执行。两者不同之处在于对信号量上加读锁。

第 351~353 行：因为地址不在地址空间中，我们检查错误是否发生于用户态。回想表 4-7，错误代码值 4 表示错误发生于用户态。

第 354~366 行：我们已经确认缺页发生在用户态，所以要发送 SIGSEGV 信号（陷入 14）。

下面的代码段是标记 `no_context:` 的流程，当跳到这点执行时，我们知道下列条件之一发生了：

- ❑ 缺少某个页表；
- ❑ 在内核态中内存访问未完成。

图 4-19 描述了标记 `no_context` 代码段的流程。

```

arch/i386/mm/fault.c
388 no_context:

390     if (fixup_exception(regs))
        return;

432     die("Oops", regs, error_code);
        bust_spinlocks(0);

```



```
do_exit(SIGKILL);
```

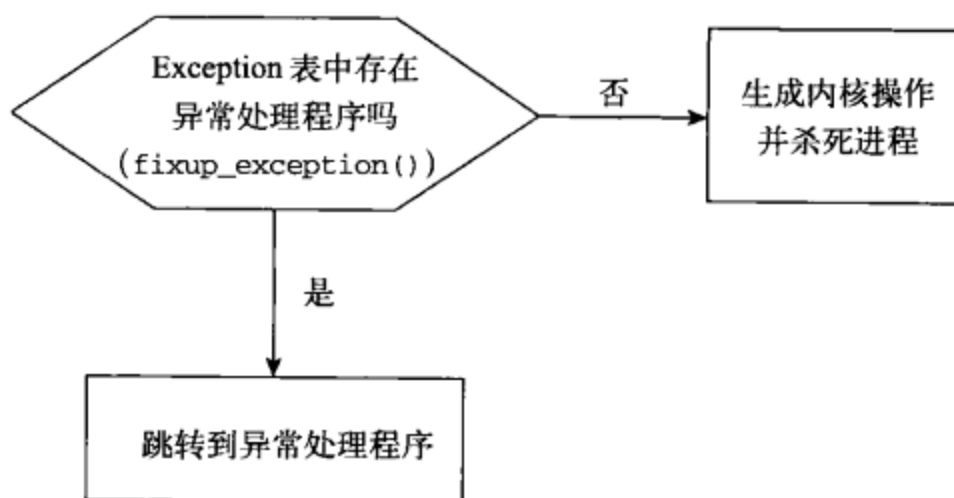


图 4-19 标记 no_context

第 390 行：函数 `fixup_exception()` 利用传入的 `eip` 参数在异常表中搜索出错的指令。如果指令存在于表中，那么说明该指令已经与内置“隐藏”的错误处理代码同被编译。缺页处理程序 `do_page_fault()` 使用错误处理代码作为返回地址，并且跳到该地址。该代码可标记一个错误。

第 432 行：如果没有在异常表中找到对应的出错指令，则跳到标记 `no_context:` 处执行，且代码以 `oops` 屏幕转储错误结束调用。

4.10.3 PowerPC 缺页异常

PowerPC 缺页处理函数 `do_page_fault()` 在指令或数据存放发生异常时被调用。因为在 PowerPC 处理器中各种版本存在细微差别，错误码有一些细微的格式不同，但是记录的信息都类似，表示产生错误的原因是读还是写，或是保护错误。PowerPC 的缺页处理程序 `do_page_fault()` 不会产生 `oops` 错误。

在 PowerPC 中，标记 `no_context` 处代码与标记 `bad_area` 处代码合并在一起，并且被置于 `bad_page_fault()` 函数中。该函数以产生段错误结束，而且该函数同样利用 `fixup` 函数检索异常表 `exception_table`。

4.11 小结

本章不但介绍与内存管理有关的所有概念，还解释了各个概念的实现。第一个概念是页，它是内核进行内存管理的基本单元。我们介绍了如何在内核中跟踪页，然后讨论了内存管理区，内存管理区受制于硬件限制，是对内存的一种划分。另外还讨论了 Linux 使用的页面分配和回收算法，这就是所谓的伙伴系统。

在讨论了页和页面管理的基本知识后，接着讨论了小于一页的小块内存分配，即由 slab 分配器管理的内存。要分配小内存块就需要借助函数 `kmalloc()` 和相关内核内存分配函数。我们跟踪了这些函数的执行过程，观察它们如何与 slab 分配器交互。到此为止，内核内存管理结构讨论完毕。

在内核管理结构和算法讨论完毕后，我们继续讨论了用户空间进程内存管理。进程内存管理不同于内核内存管理。我们讨论了进程内存布局和内存中各进程部分如何分配和映射。依照进程内存管理流程，我们介绍了缺页的概念，这个中断处理程序负责管理内存所缺少的页面。

4.12 项目：进程内存映射

现在看看我们自己程序的内存是什么样子的。该项目包括写一个用户空间程序，利用它来描述进程内存的分布。就这个项目而言，我们要创建一个简单的共享库和一个利用其函数的用户空间程序。在该程序中，我们会打印一些变量的位置，并且对比打印的位置和进程内存映射来确定变量和函数被存放的位置。

第一步我们先来创建共享库。共享库可以只有一个函数，该函数将从主程序调用。我们希望打印该函数中局部变量的地址。共享库看起来像下面这样：

```
-----
lkpsinglefoo.c
mylibfoo()
{
    int libvar;
    printf("variable libvar \t location: 0x%x\n", &libvar);
}
-----
```

编译 singlefoo.c 并将其链接成共享库：

```
#lkp>gcc -c lkpsinglefoo.c
#lkp>gcc lkpsinglefoo.o -o liblkpsinglefoo.so -shared -lc
```

-shared 和 -lc 标记是链接选项。-shared 选项要求产生可和其他对象链接在一起的共享对象，-lc 标记表示链接时需要搜索 C 库。

上述命令产生一个名为 liblkpsinglefoo.so 的文件，使用它之前需要将其复制到 /lib 目录下。

下面是我们将调用的链接在共享库中的主应用程序。

```
-----
lkpmem.c
#include <fcntl.h>

int globalvar1;
int globalvar2 = 3;

void mylocalfoo()
{
    int functionvar;
    printf("variable functionvar \t location: 0x%x\n", &functionvar);
}

int main()
{
    void *localvar1 = (void *)malloc(2048)
    printf("variable globalvar1 \t location: 0x%x\n", &globalvar1);
    printf("variable globalvar2 \t location: 0x%x\n", &globalvar2);
}
```

```

printf("variable localvar1 \t location: 0x%x\n", &localvar1);

mylibfoo();
mylocalfoo();

while(1);
return(0);
}
-----

```

按照如下命令编译 lkpmem.c:

```
#lkp>gcc -o lkpmem lkpmem.c -llkplibsingelfoo
```

执行命令 lkpmem 时,系统将打印出各种变量的内存位置。while(1); 内的函数块不被返回,以便你来得及获得进程 PID,并查看进程的内存映射。为此,要执行如下命令:

```

#lkp>./lkpmem
#lkp>ps aux | grep lkpmem
#lkp>cat /proc/<pid>/maps

```

通过上述命令你可看到每个变量所在的内存段。

4.13 习题

1. 为什么不能从一个普通的可执行文件执行进程或者为什么程序不能共享内存的数据段?
2. 下列函数在 3 次嵌套调用后,栈将如何变化?

```

foo(){
    int a;
    foo()
}

```

如果继续执行下去,会有什么问题发生?

3. 按照图 4-11 所示,填写对应于内存映射的 vm_area_struct 描述符中的相应值。
4. 页面和 slab 之间有何种关系?
5. Linux 的 32 位系统不使用页中间目录,这就是说它只有两级页表。所以虚拟地址的前 10 位对应页全局目录 (PGD) 中的偏移量,第 2 个 10 位对应页表 (PTE) 中的偏移量,剩下的 12 位对应页面内的偏移量。

Linux 系统中页面大小是多少? 一个进程可访问多少页,有多大内存可访问?

6. 内存管理区 (memory zone) 与页面之间有何种关系?
7. 从硬件角度看,“实地址”与“虚地址”有何不同?

第5章

输入/输出

本章内容

- 总线、桥、端口和接口的硬件实现
- 设备

Linux 内核可以运行于一个或多个处理器上。处理器为系统中其余部分提供的接口离不开硬件的支持。在硬件依赖性极强的底层，内核使用汇编语言指令与这些设备“对话”。本章分析内核与周围硬件之间的关系，重点放在文件 I/O 和硬件设备上，通过讨论如何将顶层虚拟文件系统的操作实现为底层向物理介质写入二进制位，来分析 Linux 内核如何将软硬件结合起来。

本章首先讨论计算机的核心——处理器，分析它如何与系统中其余部分建立连接，并讨论总线的概念，以及它们如何使处理器与系统中其他部分（例如内存）互连。同时，也介绍绝大多数 x86 系统和 PowerPC 系统都使用的构成芯片组的设备和控制器。

基本理解系统的各个组成部分及其互连后，再来分析系统中软件的层次结构，从上而下依次是：应用程序、操作系统和用于存储的特定块设备——硬盘驱动器及其控制器。虽然下一章才会提到文件系统的概念，但本章也会讨论其部分内容，以便于分析通用块设备层以及块设备最重要的通信方式：请求队列。

本章将在介绍 I/O 调度时，讨论机械设备（硬盘驱动器）和系统软件之间的重要关系。在理解硬盘驱动器的物理几何结构以及操作系统如何划分驱动器的基础上，将进一步分析软件和底层硬件之间的时间选择。

通过进一步剖析硬件，读者还将了解通用块设备驱动程序如何与特定块设备驱动程序通信，从而可以使用通用软件来控制各种硬件设备。最后，在从应用层转到 I/O 层的过程中，还将谈到磁盘控制器所需的硬件 I/O，并给出 I/O 和设备驱动器的其他例子。

接下来，还将讨论另一种主要设备类型——字符设备，以及字符设备与块设备、网络设备有何不同之处。同时，也将一并讨论 DMA 控制器、时钟设备、终端设备等与其他设备相比的重要性。

5.1 总线、桥、端口和接口的硬件实现

处理器与周围设备的通信依赖于一系列电路（或电子线路），总线（bus）就是这样一组具有

类似功能的电子线路。处理器输入/输出最常见的总线类型有：用于对设备寻址（addressing）的总线，在处理器与设备之间传送数据的总线，以及传输控制信息（例如特定于设备的初始化及其特性信息）的总线。因此，可以说设备与处理器通信（反之亦然）主要通过地址总线、数据总线和控制总线来实现。

处理器最基本的功能就是取得并执行指令。这些指令总称为**计算机程序**（computer program）或软件，程序驻留在被称之为内存的一个（或一组）设备中。处理器通过地址总线、数据总线和控制总线来访问内存。执行程序时，处理器通过地址总线来定位指令在内存中的位置，通过数据总线来传输（获取）指令，通过控制总线来确定传输方向（处理器输入/输出）和传输类型（在此，也就是内存）。当提到前端总线（front-side bus）和 PCI 总线之类的特殊总线时，是指地址总线、数据总线和控制总线的总称，这些术语可能容易引起混淆。

软件在系统中运行需要许多外围设备的支持。当前计算机系统两个主要外围设备（也称为控制器）就是**北桥**（Northbridge）和**南桥**（Southbridge）。桥一般是指连接两条总线的硬件设备。图 5-1 说明了北桥和南桥如何与其他设备互连。总之，这些控制器都是系统中的**芯片组**（chipset）。

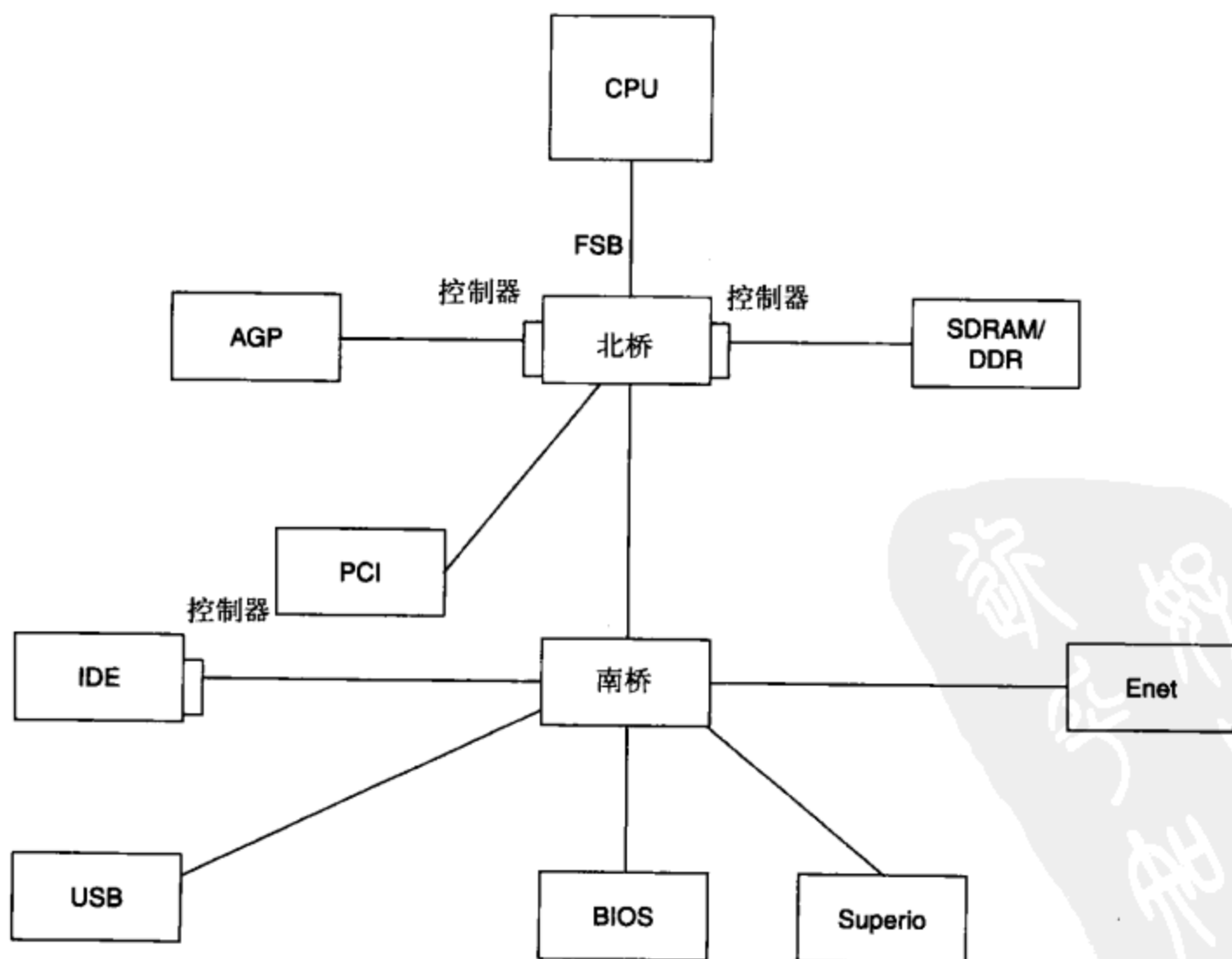


图 5-1 传统的 Intel 系统

北桥连接高速、高性能的外围设备，如内存控制器和 PCI 控制器。然而，某些芯片组在设计

之初就包含集成到北桥的图形控制器。虽然有些芯片组的设计会在北桥中集成图形控制器，但最近的设计大多会包含 AGP (Accelerated Graphics Port, 加速图形接口) 和 PCI Express 之类用来与专用图形适配器通信的高性能总线。为了获得更快的速度和良好的性能, 北桥会在前端总线^①和 (取决于特定芯片组的设计) PCI 总线或内存总线之间架起一座桥梁。

与北桥相连的南桥, 连接着一些低性能的设备。例如, Intel PIIX4 的南桥连接着 PCI-ISA 桥、IDE 控制器、USB、实时时钟、两个 82C59 中断控制器 (详细内容参见第 3 章)、82C54 定时器、两个 82C37 DMA 控制器, 以及 I/O APIC。

最早的 x86 系列个人计算机中, 与键盘、串口、并口等基本外设通讯依赖于 I/O 总线。I/O 总线是一种控制总线, 相对而言, 它与外设通信的速度较慢。x86 体系结构包含特殊的 I/O 指令, 用于在 I/O 总线上传输数据, 如 `inb` (read in a byte, 按字节读入) 和 `outb` (write out a byte, 按字节写出)。I/O 总线是通过共享处理器地址线 and 数据线来实现的。控制线只有在使用特殊的 I/O 指令时才被激活, 以免将 I/O 设备与内存混淆。PowerPC 体系结构控制外设的方式有所不同, 它采用内存映射 I/O (memory-mapped I/O) 方式给设备分配地址空间, 以便控制设备并与之通信。

例如, 在 x86 体系结构中, 第一个并口数据寄存器位于 I/O 端口 0x378 处, 而在 PPC 中, 根据其实现应在存储单元 0xf0000300 处。在 x86 中读取该寄存器的值应执行汇编指令 `in al, 0x378`, 此时为并口控制器激活了一条控制线, 告诉总线 0x378 是 I/O 端口, 而不是内存地址。在 PPC 中读取该寄存器的值则应执行汇编指令 `lbz r3, 0 (0xf0000300)`。并口控制器监视地址总线^②, 并仅当所请求的地址范围在 0xf0000300 以内时才应答。

随着个人计算机日趋成熟, 更多离散的 I/O 设备被整合成单个集成电路, 名曰超级输入输出 (Superio) 芯片。超级输入输出功能块常被进一步合并到南桥芯片 (如 ALIM1543C) 中, SMSC FDC37C932 就是将常见功能整合进离散超级设备的例子, 它包括键盘控制器、实时时钟、电源管理设备、软盘控制器、串口控制器、并口控制器、IDE 接口和通用 I/O 接口。其他南桥芯片包含集成的 LAN 控制器、PCI Express 控制器、音频控制器等。

当前, Intel 的体系结构中已经涉及集线器 (hub) 的概念。现在, 北桥被认为是 GMCH (Graphics and Memory Controller Hub, 图形和内存控制中心), 它支持高性能的 AGP 和 DDR 内存控制器。在 PCI Express 的支持下, Intel 的芯片组正向用于图形的 MCH (Memory Controller Hub, 内存控制中心) 和 DDR2 内存控制器转变。南桥就是所谓的 ICH (I/O Controller Hub, 输入/输出控制器中心)。它们通过一条名为 IHA (Intel Hub Architecture, 英特尔 Hub 架构) 的专有点对点总线互连, 详情参见 Intel 芯片组 865G^③和 925XE^④的数据表。图 5-2 给出了 ICH 的结构。

与英特尔传统的北桥/南桥风格不同, AMD 已经采用主要芯片组元件之间包化的

① 某些 PowerPC 系统中的前端总线也被称为处理器局部总线 (processor-local bus)。

② 也有人把“监视地址总线”叫做“解码地址总线”。

③ <http://www.intel.com/design/chipsets/datashts/25251405.pdf>

④ <http://www.intel.com/design/chipsets/datashts/30146403.pdf>

HyperTransport 技术。对操作系统而言,HyperTransport 与 PCI 兼容^①。详细内容请查阅 AMD 8000 系列芯片组的数据表。图 5-3 解释了 HyperTransport 技术^②。

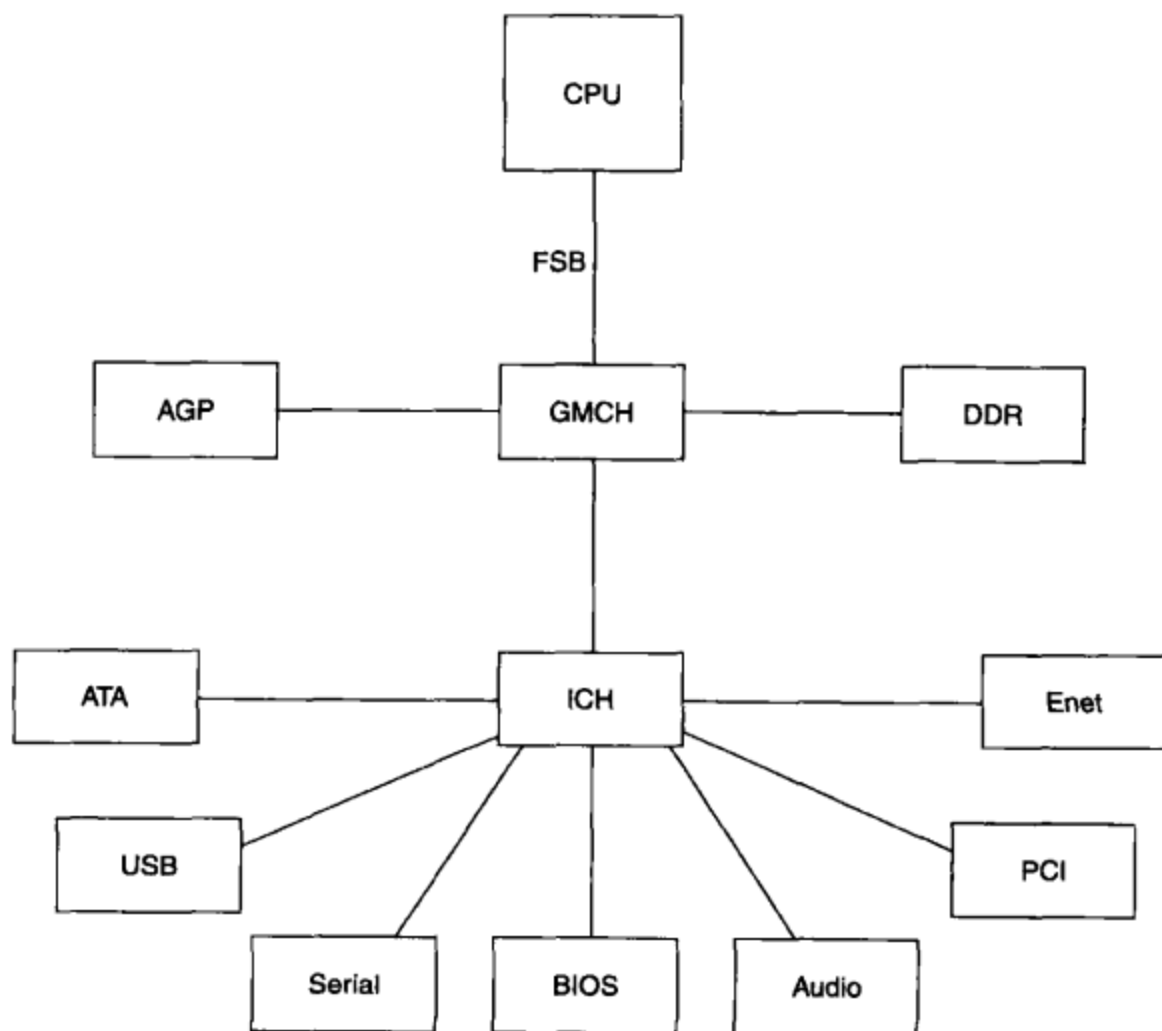


图 5-2 新的英特尔 Hub

采用 PowerPC 体系结构的苹果电脑,其专用设计叫做 UMA (Universal Motherboard Architecture, 通用主板架构)。UMA 希望能在所有 Mac 系统中使用同样的芯片组。

G4 芯片组包括北桥“UniNorth 内存控制器和 PCI 总线桥”,以及南桥“Key Largo I/O 和磁盘设备控制器”。UniNorth 支持 SDRAM、以太网和 AGP。Key Largo 南桥通过 PCI-to-PCI 桥与 UniNorth 互连,它支持 ATA 总线、USB、无线局域网 (WLAN) 和语音。

G5 芯片组包括系统控制器 ASIC (Application Specific Integrated Circuit),它支持 AGP 和 DDR 内存。PCI-X 控制器和高性能 I/O 设备通过 HyperTransport 总线与系统控制器互连。详情请参阅苹果电脑开发者专栏。

简要回顾系统的基本体系结构后,我们再来关注内核为这些设备提供的接口。第 1 章中提到过,设备也被当做文件系统中的文件来处理。和常规文件一样,这类文件也有文件权限、

① 参见 AMD 8000 系列芯片组数据表: http://www.amd.com/us-en/Processors/ProductInformation/0,30_118_6291_4886,00.html。——译者注

② HyperTransport 是一种为主板上的集成电路互连而设计的端到端总线技术,它可以在内存控制器、磁盘控制器以及 PCI 总线控制器之间提供更高的数据传输带宽。HyperTransport 采用类似 DDR 的工作方式,在 400MHz 工作频率下,相当于 800MHz 的传输频率。——译者注

文件模式以及 `open()`、`read()` 等与文件系统相关的系统调用。系统调用的重要性随着要处理的设备不同而有所不同，也可以为各种类型的设备定制系统调用。因此，设备处理的细节都可以被隐藏于内核中，而对应用级程序员透明。当进程对设备文件应用某一个系统调用时，只需将这个系统调用转换成某种在设备驱动程序中定义的设备处理函数。接下来看看设备的类型。

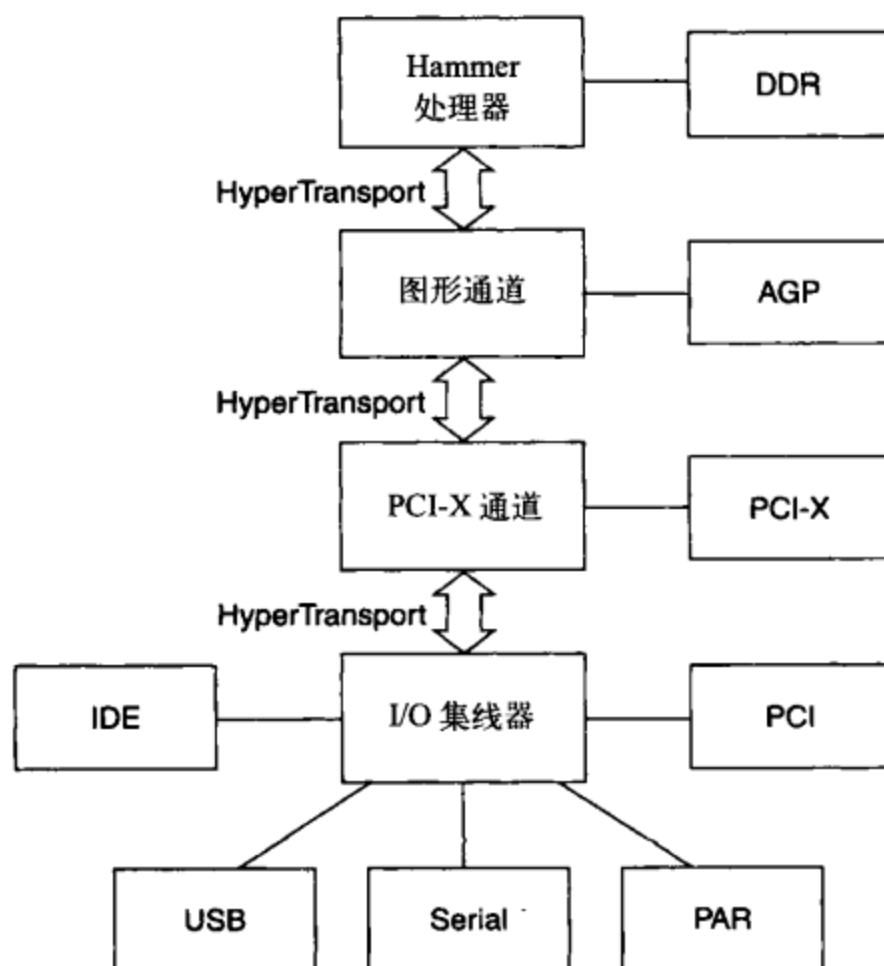


图 5-3 AMD HyperTransport

5.2 设备

系统中有两种设备文件：块设备文件和字符设备文件。块设备每次传送一块数据；顾名思义，字符设备每次只传送一个字符。第三种设备类型是网络设备，它是兼有块设备属性和字符设备属性的特殊设备，但不用文件来表示。

以前为设备分配设备号时，其主设备号通常用来表示设备驱动程序或设备控制器，从设备号则代表设备控制器所管理的特定设备。现在，这种传统方式已经被名为 `devfs` 的动态方法所取代。这种改变发生的历史原因在于主、从设备号都是 8 位数值，这使得整个系统中仅能有 200 余个静态分配的主设备（块设备和字符设备分别有自己的包含 256 个入口的列表）。`/Documentation/devices.txt` 中列出了官方已分配的主、从设备号清单。

Linux 内核自 2.3.46 版后就有了设备文件系统（`devfs`）。2.6.7 版内核中并没有默认包含 `devfs`，但可以通过在配置文件中设置 `CONFIG_DEVFS_FS=Y` 将其加入内核。有了 `devfs`，模块

就可以根据名字来注册设备，不需要再使用主/从设备号。考虑到其兼容性，devfs 允许在任何给定的系统中使用主/从设备号，或生成唯一的 16 位设备号。

5.2.1 块设备概述

如前所述，Linux 操作系统将所有设备都看做文件，可以随机引用块设备中任意给定的部分，磁盘驱动器就是一个很好的例子。第一个 IDE 磁盘在文件系统中的名字是 /dev/had，其主设备号是 3，从设备号是 0。通常，磁盘驱动器本身有一个控制器，并且本身具有机电特性（就是说，它有一些可移动部分）。第 6 章将讨论硬盘的基本结构。

通用块设备层

设备驱动程序在驱动程序初始化时注册自身，换句话说，就是将这个驱动程序加入内核的驱动程序表 (driver table) 中，并将设备号映射到数据结构 **block_device_operations** 中。block_device_operations 包含系统中启动和停止一个给定块设备的函数：

```
-----
include/linux/fs.h
760 struct block_device_operations {
761     int (*open) (struct inode *, struct file *);
762     int (*release) (struct inode *, struct file *);
763     int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
764     int (*media_changed) (struct gendisk *);
765     int (*revalidate_disk) (struct gendisk *);
766     struct module *owner;
767 };
-----
```

块设备的接口与其他设备类似，函数 open()（见第 761 行）和 release()（见第 762 行）是同步的（换言之，被调用后它们一直运行，直到任务完成）。由于自身机械特性的影响，块设备最重要的函数 read() 和 write() 的实现和常规文件有所不同。我们来分析访问磁盘上的某块数据的情况。从处理器的角度来看，将磁头定位到适当的磁道上并将磁盘转到相应的块要花费相当长的时间，这一等待时间迫使内核实现了系统请求队列 (system request queue)。当文件系统请求一块或多块不在局部页缓存 (page cache) 中的数据时，文件系统将该请求加入请求队列，并将该队列传给通用块设备层，由它来决定机械地检索（或存储）这些信息最有效的方式，然后将它们传给硬盘驱动程序。

最重要的是，初始化时，块设备驱动程序向内核（尤其是块设备管理程序）注册了一个请求队列处理程序，以方便对块设备的读写操作。通用块设备层充当文件系统和设备注册层接口之间的接口，并调优每个读队列和写队列，以便更好地使用新的智能化设备，这是通过带标记的命令队列辅助程序来实现的。例如，如果一个给定队列上的设备支持命令队列，那么，读写操作的优化就可以通过对请求重新排序来实现，使之充分利用底层硬件。此时，队列的调优就体现在能够处理多少个请求的能力上。应用层、文件系统层、通用块设备层和设备驱动程序之间的相互关系参见图 5-4。关于层的更多有用信息以及在早期版本的内核上所作改动的相关信息请参阅文件 biodoc.txt（在目录 /Documentation/block 下）。

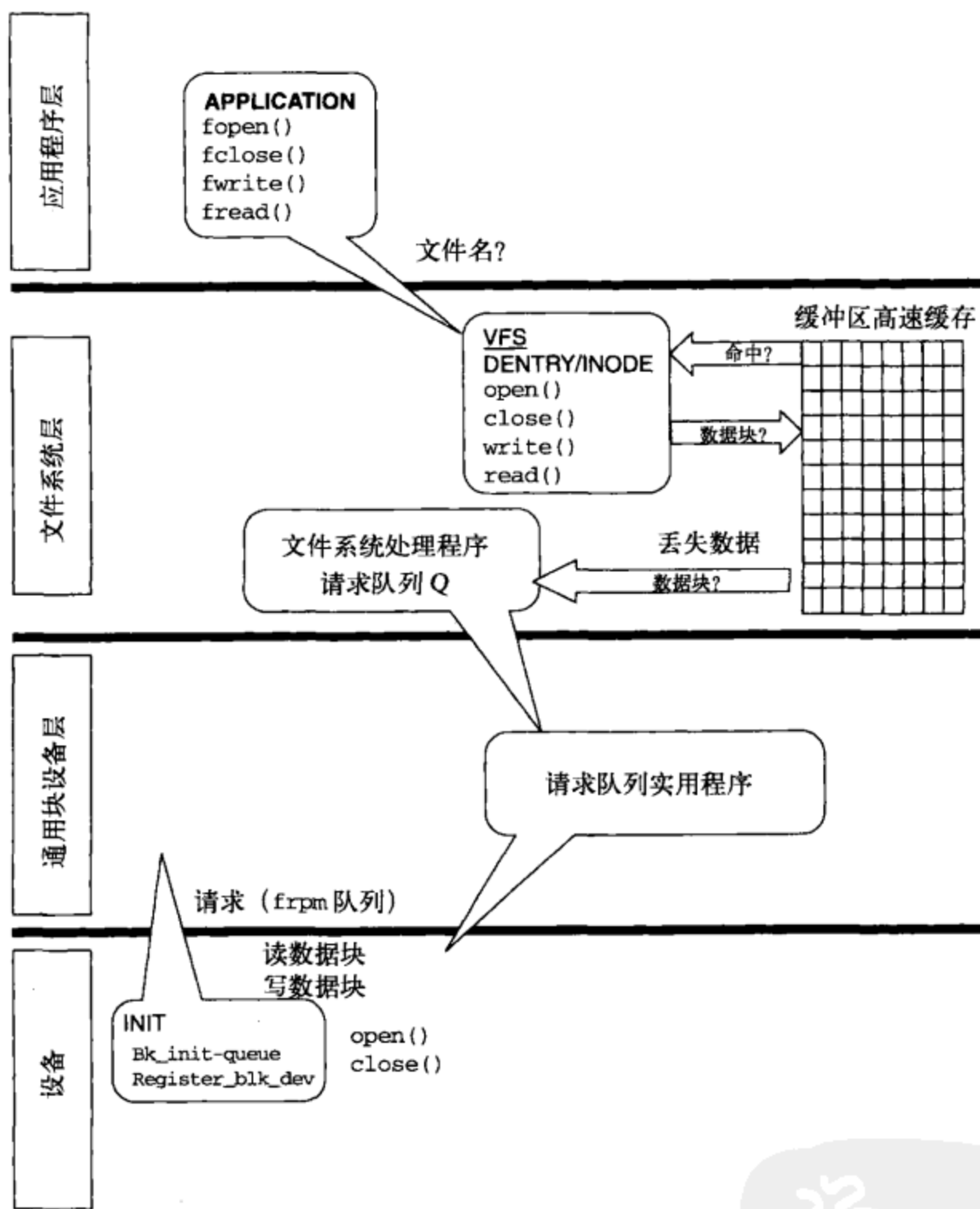


图 5-4 读/写块设备

5.2.2 请求队列和 I/O 调度

读/写请求将从 VFS 出发，通过文件系统驱动程序和页缓存^①，穿越多个层，最后到达块设备驱动程序，并执行实际的设备 I/O 操作，从而获得所请求的数据。

如前所述，块设备驱动程序在初始化时创建并初始化请求队列，同时确定读/写块设备时使用的 I/O 调度算法，即著名的**电梯调度算法 (elevator algorithm)**。

内核在启动时默认设置的 I/O 调度算法是**预期调度程序^② (anticipatory I/O scheduler)**。只要将

① 参见第 6 章。

② 某些块设备驱动程序可以在运行时更改其 I/O 调度程序——如果该调度程序在 sysfs 中可见的话。

内核参数 `elevator` 设置成下列值就可以改变 I/O 调度程序的类型。

- (1) **deadline**: 用于最后期限调度程序。
- (2) **noop**: 用于空操作调度程序。
- (3) **as**: 用于预期调度程序。

至此, 能使 I/O 调度程序完全模块化的补丁已经出现了。用户可以使用 `modprobe` 来加载模块, 以及动态地在模块之间进行切换^①, 使用这个补丁, 开始时必须至少有一个调度程序被编译到内核。

在讨论这些 I/O 调度程序如何工作之前, 还必需谈谈请求队列。

块设备使用请求队列来为给定块设备的多个 I/O 请求排序。某些块设备 (如 RAM 盘) 由于其 I/O 请求的开销很小, 可能不需要将请求排队。其他块设备 (如硬盘驱动器) 则因读写开销很大而需要将请求排队。如前所述, 硬盘驱动器的磁头必须在磁道间来回移动, 对 CPU 而言, 每次移动速度之慢简直难以忍受。

为了解决这一问题, 请求队列以优化吞吐量但并不无限推迟请求的方式将块 I/O 读写请求排队。人们常常将 I/O 调度形象地类比为电梯工作的过程^②。如果想让电梯按照提出请求的先后顺序来做出响应的話, 电梯在层与层之间移动的效率就很低, 它也可能直接从最顶层下降到最底层, 中间的各层都不停。如果允许电梯在移动时响应同一方向上的请求, 那么就可以提高运行效率, 同时还能让乘客也满意。同样, 硬盘的 I/O 请求也应该分组组织, 以免磁头来回移动带来过大的开销。前面提到的所有 I/O 调度程序 (空操作 I/O 调度、最后期限 I/O 调度和预期 I/O 调度) 都实现了基本的电梯调度功能, 下面几节将详细讨论。

1. 空操作 I/O 调度程序

空操作 I/O 调度程序^③将接收请求并扫描整个队列, 判断该请求是否能与已有的请求合并。当新请求与某个已有请求要访问的数据块相近时就可以合并。如果请求的是已有请求之前的数据块, 该请求就被合并到这个已有请求之前, 反之被合并到该已有请求之后。在一般的 I/O 操作中, 总是从头开始顺序读取文件的内容, 因此, 大部分请求均可合并到已有请求之后。

如果新请求与已有请求相距较远而不能合并, 空操作 I/O 调度程序就会在队列的已有请求之间寻找适当的位置。如果新请求调用已有请求之间的扇区, 则该请求将被插入到队列中确定的位置。如果找不到适当的位置, 该请求就会被插入到请求队列的队尾。

2. 最后期限调度程序^④

空操作调度程序存在的主要问题是, 有较多相近请求时就永远不会处理新请求。许多与已有请求相近的请求要么被合并, 要么被插入到已有请求之间, 其他新请求就会被堆积在请求队列的队尾。为了解决这个问题, 最后期限调度程序给每个请求分配一个到期时间, 并且用两个附加队列来管理有效时间, 同时用一个与空操作调度算法相似的队列来模拟磁盘效率 (disk efficiency)。

应用程序发出读请求后, 通常会一直等待, 直到该请求被满足才继续执行。但是, 如果发出

① 更多信息, 可在网上搜索 “Jens Axboe” 和 “模块化的 I/O 调度程序”。

② 这一类比正是 I/O 调度程序被称为电梯调度程序的原因。

③ 空操作 I/O 调度程序的代码可参见 `drivers/block/noop-iosched.c`。

④ 最后期限 I/O 调度程序的代码参见 `drivers/block/deadline-iosched.c`。

的是写请求，它并不会一直等到请求被满足为止。应用程序在继续处理其他任务时，写操作就可以在后台执行。因此，最后期限调度程序先响应读请求后响应写请求。除了根据所请求的扇区远近来排序的队列外，调度程序还保持读请求队列和写请求队列，而读、写请求队列中的请求根据时间先后顺序排队（FIFO）。

和空操作 I/O 调度程序一样，当新请求到来时，最后期限调度程序将它放到一个排好序的队列中，根据 I/O 请求的类别，该请求也会被加入到读请求队列或写请求队列。最后期限 I/O 调度程序处理请求时，首先检查读请求队列的队首元素，如果该请求已到期就会被马上处理。类似地，如果无任何读请求到期，调度程序就会去检查写请求队列的队首元素是否到期，如果是，马上处理该请求。仅当无任何读、写请求到期时才去检查标准队列，该队列中请求的处理方式与空操作 I/O 调度算法相似。

读请求的到期时间比写请求的到期时间短，默认时，读请求的到期时间是 0.5 秒，写请求的到期时间是 5 秒。当读请求过多时，到期时间的差异以及对读请求的优先处理可能导致写请求处于饥饿状态，因而需要一个参数通知最后期限 I/O 调度程序，读请求将导致写请求饥饿的最长时间，该参数的默认值为 2。但是，连续请求可以被当做单个请求，因而可以认为，写请求饥饿^①前能处理 32 个连续的读请求。

3. 预期 I/O 调度程序

最后期限调度算法在执行精确的写操作时存在问题。由于重点放在如何使读操作效率最高，所以写请求可以被读请求抢占，使得磁头必须重新定位，读操作完成后返回到写请求，磁头也返回原来的位置。预期 I/O 调度程序^②试图预测出下一个操作，以便提高 I/O 吞吐量。

从结构上来说，预期 I/O 调度程序与最后期限 I/O 调度程序类似，同样有根据时间排序（FIFO）的读请求队列和写请求队列，以及根据扇区远近排序的默认队列。两者的主要区别是，处理完一个读请求后，预期 I/O 调度程序并不马上处理其他请求，它在预测另一个读请求的 6 毫秒内什么都不做。如果该读请求确实发生在相邻区域，它就马上处理该请求。过了预测期后，调度程序又回到正常的操作上来，这时和最后期限调度程序中执行的操作一样。

预测期使得磁头在块设备的扇区之间来回移动带来的 I/O 延迟最小化。

与最后期限 I/O 调度程序类似，预期 I/O 调度算法中也使用了许多参数。读请求的默认到期时间是 1/8 秒，而写请求是 1/4 秒。这两个参数决定了什么时候在诸多读请求和写请求之间进行检查和切换^③。一组读请求 1/4 秒后将检查是否有到期的写请求，一组写请求 1/16 秒后将检查是否有到期的读请求。

默认的 I/O 调度程序是预期 I/O 调度程序，因为对绝大多数应用程序和块设备而言，该调度程序优化了 I/O 吞吐量。如果有数据库方面的应用或需要高磁盘性能的请求，那么使用最后期限 I/O 调度程序会更好一些。空操作 I/O 调度程序常用于 I/O 搜索时间可以忽略不计的系统，例如从 RAM 运行的嵌入式系统。

① 参数定义参见 `deadline-iosched.c` 的第 24~27 行。

② 预测 I/O 调度程序的代码参见 `drivers/block/as-iosched.c`。

③ 参数定义参见 `as-iosched.c` 中的第 30~60 行。

讨论完 Linux 内核的各种 I/O 调度程序后，我们再回过头来讨论请求队列本身以及块设备如何初始化请求队列。

4. 请求队列

在 Linux 2.6 中，每个块设备都有自己的请求队列，以便管理对该设备的 I/O 请求。进程只有获得请求队列锁后才能更新设备的请求队列。以下是 request_queue 结构：

```
-----
include/linux/blkdev.h
270 struct request_queue
271 {
272     /*
273      * Together with queue_head for cacheline sharing
274      */
275     struct list_head queue_head;
276     struct request *last_merge;
277     elevator_t elevator;
278
279     /*
280      * the queue request freelist, one for reads and one for writes
281      */
282     struct request_list rq;
-----
```

第 275 行：指向请求队列队首的指针。

第 276 行：添加到请求队列的最后一个请求。

第 277 行：调度函数（电梯调度程序）通常用于管理请求队列，它可以是某个标准 I/O 调度程序（空操作 I/O 调度程序、最后期限 I/O 调度程序或预期 I/O 调度程序），或为块设备定制的某种新类型的调度程序。

第 282 行：request_list 数据结构由两个 wait_queue 组成，分别用于块设备读请求队列和写请求队列。

```
-----
include/linux/blkdev.h
283
284 request_fn_proc *request_fn;
285 merge_request_fn *back_merge_fn;
286 merge_request_fn *front_merge_fn;
287 merge_requests_fn *merge_requests_fn;
288 make_request_fn *make_request_fn;
289 prep_rq_fn *prep_rq_fn;
290 unplug_fn *unplug_fn;
291 merge_bvec_fn *merge_bvec_fn;
292 activity_fn *activity_fn;
293
-----
```

第 283~293 行：定义与调度程序（或电梯调度程序）相关的函数，以控制如何来管理块设备的请求。

```
-----
include/linux/blkdev.h
```

```

294  /*
295  * Auto-unplugging state
296  */
297  struct timer_list unplug_timer;
298  int unplug_thresh; /* After this many requests */
299  unsigned long unplug_delay; /* After this many jiffies*/
300  struct work_struct unplug_work;
301
302  struct backing_dev_info backing_dev_info;
303
-----

```

第 294~303 行：这些函数常用于“拔掉”块设备的 I/O 调度函数。插入 (plugging) 是指等待更多请求加入请求队列，同时期待更多请求允许调度算法对其排序，以减少执行 I/O 请求所花的时间。例如，硬盘驱动器“插入”一定数目的读请求，期待存在更多读请求时磁头移动较少。这更像是读请求可以顺序排列，或组合成单个大的读请求。拔下 (unplugging) 是指设备决定不再等待并使得系统必须响应其请求的一种方式，而不考虑将来可能的优化。详情可查阅 [documentation/block/biodoc.txt](#)。

```

-----
include/linux/blkdev.h
304  /*
305  * The queue owner gets to use this for whatever they like.
306  * ll_rw_blk doesn't touch it.
307  */
308  void *queuedata;
309
310  void *activity_data;
311
-----

```

第 304~311 行：正如内部注释所说的那样，这些行用于请求特定于设备及其设备驱动程序的队列管理。

```

-----
include/linux/blkdev.h
312  /*
313  * queue needs bounce pages for pages above this limit
314  */
315  unsigned long bounce_pfn;
316  int bounce_gfp;
317
-----

```

第 312~317 行：跳跃 (bouncing) 是指内核将高端内存缓冲区的 I/O 请求复制到低端内存缓冲区。在 Linux 2.6 中，内核允许设备在需要时管理高端内存缓冲区。跳跃仅当设备不能处理高端内存缓冲区时才会发生。

```

-----
include/linux/blkdev.h
318  /*
319  * various queue flags, see QUEUE_* below

```

```

320  */
321  unsigned long   queue_flags;
322
-----

```

第318~321行：变量 `queue_flag` 存储一个或多个队列标志，参见表 5-1（参阅 `include/linux/blkdev.h` 的第 368~375 行）。

表5-1 `queue_flags`

标 志 名	功 能
<code>QUEUE_FLAG_CLUSTER</code>	将几个段合并成一段
<code>QUEUE_FLAG_QUEUED</code>	使用通用标记来排队
<code>QUEUE_FLAG_STOPPED</code>	队列被停止
<code>QUEUE_FLAG_READFULL</code>	读队列已满
<code>QUEUE_FLAG_WRITEFULL</code>	写队列已满
<code>QUEUE_FLAG_DEAD</code>	队列被撤销
<code>QUEUE_FLAG_REENTER</code>	避免重入
<code>QUEUE_FLAG_PLUGGED</code>	插入队列

```

-----
include/linux/blkdev.h
323  /*
324  * protects queue structures from reentrancy
325  */
326  spinlock_t    *queue_lock;
327
328  /*
329  * queue kobject
330  */
331  struct kobject kobj;
332
333  /*
334  * queue settings
335  */
336  unsigned long   nr_requests; /* Max # of requests */
337  unsigned int    nr_congestion_on;
338  unsigned int    nr_congestion_off;
339
340  unsigned short   max_sectors;
341  unsigned short   max_phys_segments;
342  unsigned short   max_hw_segments;
343  unsigned short   hardsect_size;
344  unsigned int     max_segment_size;
345
346  unsigned long    seg_boundary_mask;
347  unsigned int     dma_alignment;
348
349  struct blk_queue_tag *queue_tags;
350
351  atomic_t         refcnt;

```

```

352
353 unsigned int    in_flight;
354
355 /*
356  * sg stuff
357  */
358 unsigned int    sg_timeout;
359 unsigned int    sg_reserved_size;
360 };

```

第 323~360 行：这些变量定义了请求队列中可管理的资源，如锁（第 326 行）和内核对象（第 331 行），同时还提供了特殊的请求队列设置，如最大请求数（第 336 行）和块设备的物理约束（第 340~347 行）。如果块设备能使用 SCSI，还可以定义 SCSI 属性（第 355~359 行）。如果想要使用标记的命令队列，则可使用数据结构 `queue_tags`（第 349 行）来实现。`refcnt` 和 `in_flight`（第 351 行和第 353 行）用于计算队列的引用数（通常用于锁）和进程中的动态请求数。

Linux 2.6 内核通过在设备的 `__init` 函数中调用下列函数来初始化块设备的请求队列。在这些函数中，可以看到请求队列内部的细节和相关的辅助例程。相对早期的 Linux 而言，在 Linux 2.6 内核中，每个块设备控制自己的锁，并将自旋锁作为第二个参数传递。第一个参数是块设备驱动程序提供的请求函数。

```

-----
drivers/block/ll_rw_blk.c
1397 request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
1398 {
1399     request_queue_t *q;
1400     static int printed;
1401
1402     q = blk_alloc_queue(GFP_KERNEL);
1403     if (!q)
1404         return NULL;
1405
1406     if (blk_init_free_list(q))
1407         goto out_init;
1408
1409     if (!printed) {
1410         printed = 1;
1411         printk("Using %s io scheduler\n", chosen_elevator->elevator_name);
1412     }
1413
1414     if (elevator_init(q, chosen_elevator))
1415         goto out_elv;
1416
1417     q->request_fn    = rfn;
1418     q->back_merge_fn  = ll_back_merge_fn;
1419     q->front_merge_fn = ll_front_merge_fn;
1420     q->merge_requests_fn = ll_merge_requests_fn;
1421     q->prep_rq_fn     = NULL;
1422     q->unplug_fn      = generic_unplug_device;
1423     q->queue_flags    = (1 << QUEUE_FLAG_CLUSTER);
1424     q->queue_lock     = lock;

```

```

1425
1426 blk_queue_segment_boundary(q, 0xffffffff);
1427
1428 blk_queue_make_request(q, __make_request);
1429 blk_queue_max_segment_size(q, MAX_SEGMENT_SIZE);
1430
1431 blk_queue_max_hw_segments(q, MAX_HW_SEGMENTS);
1432 blk_queue_max_phys_segments(q, MAX_PHYS_SEGMENTS);
1433
1434 return q;
1435 out_elv:
1436 blk_cleanup_queue(q);
1437 out_init:
1438 kmem_cache_free(requestq_cachep, q);
1439 return NULL;
1440 }

```

第 1402 行：从内核的内存为队列分配空间，并将其内容初始化为 0。

第 1406 行：初始化包含读队列和写队列的请求清单。

第 1414 行：将所选择的电梯调度程序与该队列关联并初始化。

第 1417~1424 行：将与电梯调度程序相关的函数与该队列关联。

第 1426 行：该函数为段归并设置边界，并检查它是否满足最小尺寸。

第 1428 行：该函数设置驱动程序用于从队列删除请求的函数。它允许使用其他函数来绕过该队列。

第 1429 行：初始化归并段的上限。

第 1431 行：初始化物理设备可处理的最大段数。

第 1432 行：初始化每一请求的最大物理段数目。第 1429~1432 行设置该值，详细内容可参阅 `include/linux/blkdev.h`。

第 1434 行：返回已初始化的队列。

第 1435~1439 行：这是错误事件中用于清除内存的例程。

现在，已经有了位于适当位置并已初始化的请求队列。

在分析通用设备层和通用块设备驱动程序之前，先来简要回顾处理块设备 I/O 操作的软件层（参见图 5-4）。

在应用层，应用程序初始化 `fread()` 等文件操作，虚拟文件系统（VFS）层将接收这一请求（详情请参阅第 4 章）。它保存文件的 `dentry` 结构，并通过 `inode` 结构调用文件的 `read()` 函数。VFS 层试图在缓冲区高速缓存中寻找请求的页面，如果该页面不存在，就调用文件系统处理程序来获取适当的物理块。`inode` 链接到文件系统处理程序，该处理程序又和某一文件系统相关。文件系统处理程序调用请求队列实用程序，是通用块设备层的一部分）为相应物理块和设备创建请求，这一请求被加入请求队列，通用块设备层将维护这个队列。

5.2.3 示例：“通用”块设备驱动程序

我们首先来分析通用块设备层。如图 5-4 所示，通用块设备层位于物理设备层之上、文件系

统层之下。它最重要的工作就是维护请求队列及其相关的辅助例程。

注册设备时使用函数 `register_blkdev(major, dev_name, fops)`。该函数接收被请求的主设备号、块设备名（在 `/dev` 目录下）和指向数据结构 `file operations` 的指针，注册成功后将返回所请求的主设备号。

接下来创建 `gendisk` 数据结构。

头文件 `include/linux/genhd.h` 中定义了函数 `alloc_disk(int minors)`，该函数接收分区号并返回指向数据结构 `gendisk` 的指针。数据结构 `gendisk` 的内容如下：

```
-----
include/linux/genhd.h
081 struct gendisk {
082     int major;      /* major number of driver */
083     int first_minor;
084     int minors;
085     char disk_name[16]; /* name of major driver */
086     struct hd_struct **part; /* [indexed by minor] */
087     struct block_device_operations *fops;
088     struct request_queue *queue;
089     void *private_data;
090     sector_t capacity;
091
092     int flags;
093     char devfs_name[64]; /* devfs crap */
094     int number; /* more of the same */
095     struct device *driverfs_dev;
096     struct kobject kobj;
097
098     struct timer_rand_state *random;
099     int policy;
100
101     unsigned sync_io; /* RAID */
102     unsigned long stamp, stamp_idle;
103     int in_flight;
104 #ifdef CONFIG_SMP
105     struct disk_stats *dkstats;
106 #else
107     struct disk_stats dkstats;
108 #endif
109 };
-----
```

第 82 行：将函数 `register_blkdev()` 的返回值写入 `major_num` 字段。

第 83 行：硬盘驱动器的块设备能够管理几个物理驱动器。虽然块设备依赖于驱动程序，但其从设备号通常用来标识每一个物理驱动器。`first_minor` 是第一个物理驱动器的从设备号。

第 85 行：`hda` 和 `sdb` 之类的磁盘名 (`disk_name`) 是整个磁盘的文本名称（磁盘上的分区分别叫做 `hda1`, `hda2`, 依次类推），是物理磁盘设备上的逻辑磁盘。

第 87 行：`fops` 字段是已初始化为 `file operations` 数据结构的 `block_device_operations`。`file operations` 包含了底层设备驱动程序中指向辅助函数的指针。由于并不是

所有驱动程序中都实现了这些函数，因此这些函数依赖于特定的驱动程序。普遍实现了的文件操作有 open、close、read 和 write。第4章已讨论过 file operations 数据结构。

第 88 行：queue 字段指向驱动程序必须执行的请求操作列表。稍后将讨论请求队列的初始化。

第 89 行：private_data 指针指向那些依赖于特定驱动程序的数据。

第 90 行：capacity 被设置成驱动器的大小（单位是 1 个扇区，512 KB）。调用函数 set_capacity() 可以获得这个数值。

第 92 行：flags 用于标明设备的属性。如果是磁盘驱动器，flags 存储的就是介质类型，例如 CD，可移动介质，等等。

现在，我们来考虑初始化请求队列时要涉及哪些内容。由于该队列已声明，因此，只需调用函数 blk_init_queue(request_fn_proc, spinlock_t) 即可。该函数把代表文件系统的转换函数作为它的第一个参数，并调用 blk_alloc_queue() 来分配队列，同时初始化该队列结构。对所有操作而言，blk_init_queue() 的第二个参数是与队列相关的锁。

最后，要使这个块设备对内核可见，驱动程序还必须调用函数 add_disk()：

```
-----
Drivers/block/genhd.c
193 void add_disk(struct gendisk *disk)
194 {
195     disk->flags |= GENHD_FL_UP;
196     blk_register_region(MKDEV(disk->major, disk->first_minor),
197         disk->minors, NULL, exact_match, exact_lock, disk);
198     register_disk(disk);
199     blk_register_queue(disk);
200 }
-----
```

第 196 行：根据其大小和分区号将设备映射到内核。

调用函数 blk_register_region() 时需要如下 6 个参数。

- 磁盘的主设备号和第一个从设备号。
- 第一个从设备号之后的从设备号范围（如果这个驱动程序管理多个从设备的话）。
- 包含驱动程序的可加载模块（如果有驱动程序的话）。
- 用于查找相应磁盘的例程 exact_match。
- 加锁函数 exact_lock。exact_match 例程找到相应的磁盘以后，该函数就给这些代码加锁。
- 处理程序 disk，用于帮助 exact_match 例程和 exact_lock 函数识别特定磁盘。

第 198 行：register_disk 检查磁盘分区并将其加入文件系统。

第 199 行：为特定区域注册请求队列。

5.2.4 设备操作

基本的通用块设备有 open、close（或 release）和 ioctl 函数，以及最重要的 request 函数。至少，open 和 close 可以单纯作为计数器来使用。ioctl() 接口可以绕过软件层来进行调试和性能测试。文件系统将一个请求加入队列时将调用 request 函数，并抽取出其数据结构，

按照该结构的具体内容进行操作。设备根据该请求是读还是写来执行相应的操作。

对请求队列的访问只能通过一组辅助例程来实现（可参阅 `drivers/block/elevator.c` 和 `include/linux/blkdev.h`）。为了与基本设备模型保持一致，应该在 `request` 函数中加入作用于下一个请求的功能：

```
-----
drivers/block/elevator.c
186 struct request *elv_next_request(request_queue_t *q)
-----
```

这个辅助函数返回指向下一个请求的数据结构的指针，驱动程序可以通过查看其内容来收集所有信息，以便确定其大小、方向以及与该请求相关的任何其他自定义操作。

驱动程序完成该请求后，将通过辅助函数 `end_request()` 向内核报告这一信息：

```
-----
drivers/block/ll_rw_blk.c
2599 void end_request(struct request *req, int uptodate)
2600 {
2601     if (!end_that_request_first(req, uptodate, req->hard_cur_sectors)) {
2602         add_disk_randomness(req->rq_disk);
2603         blkdev_dequeue_request(req);
2604         end_that_request_last(req);
2605     }
2606 }
-----
```

第 2599 行：在请求队列中传递从函数 `elv_next_request()` 获得的参数。

第 2601 行：函数 `end_that_request_first()` 用于传递扇区号（如果扇区尚未确定，则函数 `end_request()` 直接返回）。

第 2602 行：加入系统熵池。熵池是一个系统方法，中断时被调用，从而从函数快速产生随机数，其基本思想是从系统内各种驱动程序中收集数据，并以此来生成随机数。第 10 章将讨论这个问题，文件 `drivers/char/random.c` 的开头部分对此也有解释。

第 2603 行：从队列中删除请求数据结构。

第 2604 行：收集统计信息并释放可用的数据结构。

由此看来，通用驱动程序服务所有请求，直到它被释放为止。

参阅图 5-4，现在，已经有了创建并维护请求队列的通用块设备层。块设备 I/O 系统的最后一层是硬件（或特殊的）设备驱动程序。硬件设备驱动程序在通用设备驱动层使用请求队列辅助例程，来响应已注册的请求队列中的请求，并在完成该请求后发送通知。

硬件设备驱动程序包含与寄存器位置、I/O、定时、中断和 DMA（参见 5.2.9 节）等相关的底层硬件的详细信息。本章并不讨论完整的 IDE 或 SCSI 驱动程序有多复杂，第 10 章将提供更多硬件设备驱动程序的信息，以及一组可帮助你建立驱动程序框架的项目。

5.2.5 字符设备

与块设备不同，字符设备用于传送数据流。所有串行设备都是字符设备。键盘控制器和串行

终端就是典型的字符流设备，使用这些设备时，很明显不能（我们也不想）从中随机访问数据。这就为分组数据传输引入了空白。以太网在物理传输层的介质是串行设备，但在总线级，它将使用 DMA 方式在存储器与外设之间传输数据块。

设备驱动程序设计人员可以让硬件做任何事情，但实时性和实用性才是决定性的因素，因而人们无法随机访问 IDE 驱动器上的音频流和流数据。虽然这两种想法看起来都很有吸引力，但还是必须遵守以下两个简单的规则。

- 所有 Linux 设备 I/O 都基于文件。

- 所有 Linux 设备 I/O 要么是字符设备 I/O，要么是块设备 I/O。

本章最后讨论的并口驱动程序是一个字符设备驱动程序。字符设备驱动程序和块设备驱动程序的相似之处在于：它们都是基于文件 I/O 的接口。表面上，两者都使用 open、close、read 和 write 等文件操作。实质上，它们之间最显著的区别就是，字符设备没有读写操作的请求队列的块设备系统（前面已经讨论过）。对没有缓冲区的字符设备而言，每接收到一个元素（字符）就产生一个中断。块设备则在接收一个数据块后才产生一个中断。

5.2.6 网络设备

网络设备兼有块设备和字符设备的一些属性，通常被认为是一组特殊的设备。与字符设备类似，网络设备的数据在物理层上串行传输。但数据包化并通过 DMA（direct memory access，直接存储器存取）方式在网络控制器上传输（参见 5.2.9 节），这一点与块设备相似。

作为 I/O 设备的一种，本章也提到了网络设备，但鉴于其过于复杂，本书不详细讨论。

5.2.7 时钟设备

时钟是用于对系统的硬件检测信号（heartbeat）计数的 I/O 设备。如果没有时间流逝的概念，Linux 就岿然不动了。第 7 章将介绍系统时钟和实时时钟。

5.2.8 终端设备

最早的终端设备是电传打字机（因而串口驱动程序被称为 tty）。自从世纪之交人们希望在电报线路上传送实时文本数据以来，电传打字机已经有了极大发展。20 世纪 60 年代初，电传打字机随着早期 RS-232 标准的诞生而日趋成熟，并似乎跟得上不断增长的小型机的发展脚步。为了与计算机进行通信，20 世纪 70 年代，电传打字机不得不让位于计算机终端。真正意义上的计算机终端却是凤毛麟角，从 20 世纪 70 年代到 80 年代中期，随着大型机和微型机的风行，它们又被运行终端仿真软件包的 PC 所取代。终端本身（也称为“哑”终端）只是显示器和通过串行方式与大型机通信的键盘设备。与 PC 不同的是，它的“智能”仅限于发送和接收文本数据。

主控制台（可在引导时配置）是 Linux 系统启动后运行的第一个终端，通常会加载一个图形界面，之后是终端仿真程序窗口。

5.2.9 直接存储器存取

DMA（Direct Memory Access，直接存储器存取）控制器是位于 I/O 设备和（通常是）系统

中高性能总线之间的硬件设备。它的主要作用是在无处理器干预的情况下传输大量数据。DMA 控制器可以看成是专用处理器，可以对它编程以实现和主存之间的大块数据传输。在寄存器级，DMA 控制器只要获得源地址、目的地址和要传送的数据长度就能够完成数据传送任务。然后，当主处理器空闲时，它就可以处理大量数据的传送（从设备到存储器、从存储器到存储器或从存储器到设备）。

许多控制器（例如磁盘控制器，网络控制器和图形控制器）都有内置的 DMA 引擎，因此可以实现大量数据传输而不占用宝贵的 CPU 周期。

5.3 小结

本章描述 Linux 内核如何处理输入和输出操作，更准确地说，讨论了如下几点：

- 简要回顾了 Linux 内核用于执行底层输入输出操作的硬件，如桥和总线；
- 讨论了 Linux 如何表示块设备及如何与块设备交互；
- 介绍了各种 Linux 调度程序和请求队列：空操作 I/O 调度程序、最后期限 I/O 调度程序、预期 I/O 调度程序。

5.4 项目：创建并口驱动程序

本项目简要介绍并口控制器，举例说明前面讨论过的 I/O 例程如何合并。并口通常被集成到芯片组的超级输入输出中，是字符设备驱动程序框架的一个很好的例子。该驱动程序（或可动态加载的模块）并不是非常有用，但可以在此基础上进一步组建并完善其功能。由于是在寄存器级实现对设备的寻址，因此，只要记录了寄存器 I/O 映射，该模块就可以用于访问 PowerPC 系统的 I/O。

本项目的并口设备驱动程序使用标准函数 `open()`、`close()`，最重要的是使用 `ioctl()` 接口来说明该设备驱动程序的结构及其内部工作原理。本项目中并不使用 `read()` 和 `write()` 函数，因为 `ioctl()` 调用已经返回了寄存器的值（因为该设备驱动程序是一个可动态加载的模块，此处仅把它当做一个模块来处理）。

首先，本项目将简要描述如何与并口对话，然后深入研究字符设备驱动程序模块的基本操作。对设备中寄存器的引用通过 `ioctl()` 接口来实现，同时也创建应用程序与该模块进行交互。

5.4.1 并口的硬件

在网上任意搜索“并口”都能得到大量信息。本节的目的是描述 Linux 的模块，因而仅提及这一设备的基本概念。

本项目的实验环境是 x86 系统，驱动程序的框架可以很容易地移植到 PowerPC 中，仅需在 I/O 层与另一设备对话即可。虽然很多嵌入式 PowerPC 系统中都提供了并口，但它并未广泛用于桌面系统（如 G4 和 G5）。

为了真正与并口寄存器通信，要使用 `inb()` 和 `outb()`，它们就像 `readb()` 和 `writb()` 一样易于操作。在 x86 和 PowerPC 中，这两条指令都是在头文件 `io.h` 中定义的。宏 `readb()` 和 `writb()` 都可以处理 x86 和 PowerPC 中的底层 I/O 例程，因而与体系结构无关。

x86 系统中的并口常被当做超级输入输出设备的一部分，或者是添加到系统中的单个 (PCI) 卡。如果进入系统的 BIOS 设置程序，可以看到并口被映射到系统 I/O 地址空间的哪个位置。对 x86 系统而言，并口位于 16 进制地址 0x278、0x378 或 0x3bc 处，并使用中断 IRQ7。这是设备的基地址 (base address)。并口有 3 个 8 位寄存器，其基地址如表 5-2 所示。本例中使用 0x378。

表5-2 并口寄存器

位	7	6	5	4	3	2	1	0	I/O端口地址
数据寄存器 (输出)	D7	D6	D5	D4	D3	D2	D1	D0	0x378 (base+0)
状态寄存器 (输入)	Busy*	ACK	Paper end	Select	Error				0x379 (base+1)
控制寄存器 (输出)					Select*	Init	Auto feed*	Strobe*	0x37A (base+2)

*低有效

数据寄存器存放可写出到连接器针脚的 8 位信息。

状态寄存器存放从连接器输入的信号。

控制寄存器向连接器发送特定的控制信息。

并口连接器是 25 针的 D-shell (DB-25)。表 5-3 列出了这些信号如何映射到连接器的特定针脚。

表5-3 信号与并口连接器针脚之间的关系

信号名	针脚号
Strobe	1
D0	2
D1	3
D2	4
D3	5
D4	6
D5	7
D6	8
D7	9
Acknowledge	10
Busy	11
Paper end	12
Select in	13
Auto feed	14
Error	15
Initialize	16
Select	17
Ground	18~25

警告 并口对静电和过电流都很敏感，不要使用集成的（嵌入主板的）并口，除非：

☐ 你对自己的硬件技能很有信心；

☐ 你确信不会毁坏端口和主板。

强烈建议你在本次实验，甚至所有实验中都使用并口适配器。

执行输入操作时，用 470Ω 电阻将 D7（针 9）跨接到 Acknowledge（针 10），D6（针 8）跨接到 Busy（针 11）。为了监视输出，可以用 470Ω 的限流电阻驱动 LED 和数据针 D0~D4，可以在本地使用老式的打印线缆或 25 针凸模 D-shell 连接器来实现这一功能。

注意 一位优秀的寄存器级程序员应该知道尽可能多的底层硬件知识，包括为特殊并口 I/O 设备找出数据表。在数据表中，你可以找到设备的反向/源电流限制。许多网站都提供了并口的接口方法，包括隔离、扩展信号数和增大/减小电阻。任何想了解 I/O 控制器如何工作的人都必须看看这些内容，但本例并不讨论这些。

本模块通过函数 `outb()` 和 `inb()` 来实现并口寻址。第 2 章曾提到这些函数依赖于平台编译环境，在 x86 系统中可以正确地实现 `in` 和 `out` 指令，在采用内存映射的 I/O 方式的 PowerPC 系统中则是 `lbz` 和 `stb` 指令。可以在头文件 `io.h` 中找到这两个平台的内联代码。

5.4.2 运行在并口上的软件

下面讨论与本项目相关的驱动程序函数。`parll.c`、`Make` 和 `parll.h` 文件完整的程序清单，参见书末。

1. 设置文件操作（fops）

如前所述，该模块使用 `open()`、`close()`、`ioctl()` 以及在前面的项目中讨论过的 `init` 和 `cleanup` 操作。

第一步是设置 `file operations` 数据结构。文件 `/linux/fs.h` 中定义了该数据结构，它列举出所有可能用到的函数，可以选择合适的函数在该模块中来实现。当然，只要实现那些需要用到的操作就可以了，而不是详细列出每一个操作。在网上搜索 C99 和 Linux 模块（Linux module）可以获得这一技术的很多信息。使用该数据结构可将 `open`、`release` 和 `ioctl` 的实现（或入口点）位置告知内核。

```
-----
parll.c
struct file_operations parlport_fops = {
    .open =    parlport_open,
    .ioctl =   parlport_ioctl,
    .release = parlport_close };
-----
```

接下来，就是创建函数 `open()` 和 `close()`。这些函数实际上是虚构的，只是在打开和关闭文件时提供一个标志。

parll.c

```
static int parlport_open(struct inode *ino, struct file *filp)
{
    printk("\n parlport open function");
    return 0;
}

static int parlport_close(struct inode *ino, struct file *filp)
{
    printk("\n parlport close function");
    return 0;
}

-----
```

然后创建 `ioctl()` 函数。注意，以下声明应该放在文件 `parll.c` 的开头：

#define MODULE_NAME "parll"
static int base = 0x378;

parll.c

```
static int parlport_ioctl(struct inode *ino, struct file *filp,
    unsigned int ioctl_cmd, unsigned long parm)
{
    printk("\n parlport ioctl function");
    if(_IOC_TYPE(ioctl_cmd) != IOCTL_TYPE)
    {
        printk("\n%s wrong ioctl type",MODULE_NAME);
        return -1;
    }
    switch(ioctl_cmd)
    {
        case DATA_OUT:
            printk("\n%s ioctl data out=%x",MODULE_NAME,(unsigned int)parm);
            outb(parm & 0xff, base+0);
            return (parm & 0xff);

        case GET_STATUS:
            parm = inb(base+1);
            printk("\n%s ioctl get status=%x",MODULE_NAME,(unsigned int)parm);
            return parm;

        case CTRL_OUT:
            printk("\n%s ioctl ctrl out=%x",MODULE_NAME,(unsigned int)parm);
            outb(parm && 0xff, base+2);
            return 0;

    } //end switch
    return 0;
} //end ioctl

-----
```

函数 `ioctl()` 用于处理用户自定义的命令。本模块向用户提供和并口相关的 3 个寄存器。`DATA_OUT` 命令发送一个数据给数据寄存器 (data register), `GET_STATUS` 命令从状态寄存器 (status register) 读取数据, 最后, `CTRL_OUT` 命令设置端口的控制信号。虽然更好的方式是隐藏 `read()` 和 `write()` 例程后的设备细节, 但该模块主要是为了实验 I/O 操作, 而不是封装数据。

这三个命令都已经在头文件 `parll.h` 中定义好了, 创建时通过使用 `IOCTL` 辅助例程作类型检查来实现。我们使用 `IOCTL` 类型检查宏 `IO (type, number)` 而不是一个整数来表示 `IOCTL` 函数, 这里 `type` 定义为 `p` (并口的类型), `number` 则是 `case` 语句中使用的实际 `IOCTL` 数。在 `parlport_ioctl()` 的开头, 要检查类型是否为 `p`。由于应用程序代码使用相同的头文件作为驱动程序的头文件, 因而接口能保持一致。

2. 设置模块初始化例程

初始化模块将该模块与操作系统联系起来, 需要时也可以用于提前初始化任何数据结构。由于这个并口驱动程序不需要用到复杂的数据结构, 因此只注册了该模块。

```
-----
parll.c
static int parl_init(void)
{
    int retval;

    retval= register_chrdev(Major, MODULE_NAME, &parlport_fops);
    if(retval < 0)
    {
        printk("\n%s: can't register",MODULE_NAME);
        return retval;
    }
    else
    {
        Major=retval;
        printk("\n%s:registered, Major=%d",MODULE_NAME,Major);

        if(request_region(base,3,MODULE_NAME))
            printk("\n%s:I/O region busy.",MODULE_NAME);

    }
    return 0;
}
-----
```

函数 `init_module()` 负责向内核注册该模块。函数 `register_chrdev()` 接收请求的主设备号 (5.2 节已经讨论过, 稍候第 10 章还会再次讨论。若主设备号为 0, 则内核将为该模块动态分配一个号)。回想一下, 主设备号是保存在 `inode` 数据结构中, 数据结构 `dentry` 指向 `inode`, 而指向 `dentry` 的指针则存储在一个文件结构中。第二个参数是出现在 `/proc/device` 中的设备名。第三个参数就是上面提到的 `file operations` 数据结构。

成功注册模块之前, `init` 例程用并口基地址和将要操作的寄存器范围的长度 (单位为字节) 来调用函数 `request_region()`。

如果注册失败, 函数 `init_module()` 就会返回一个负数。

3. 设置模块的清理例程

函数 `cleanup_module()` 负责注销模块，并释放先前请求的 I/O 范围：

```
-----
par11.c

static void par11_cleanup( void )
{
    printk("\n%s:cleanup ",MODULE_NAME);
    release_region(base,3);
    unregister_chrdev(Major,MODULE_NAME);
}
-----
```

最后，将所需的 `init` 和 `cleanup` 入口点包括进来：

```
-----
par11.c
module_init(par11_init);
module_exit(par11_cleanup);
-----
```

4. 插入模块

现在可以像前一个项目那样将模块插入到内核中了，所用命令如下：

```
Lkp: ~# insmod par11.ko
```

查看 `/var/log/messages`，发现 `init()` 例程的输出就像前面所说的那样，但返回了更详细的主设备号信息。

前一个项目仅向内核插入模块并从内核移除该模块。现在，需要使用命令 `mknod` 将模块与文件系统联系起来。在命令行输入如下命令：

```
Lkp: ~# mknod /dev/par11 c <xxx> 0
```

其中参数说明如下。

- **c**：创建特殊的字符设备文件（与块设备文件相对）。
- **/dev/par11**：到该设备的路径（供 `open` 调用使用）。
- **xxx**：执行初始化时返回的主设备号（从 `/var/log/messages` 返回）。
- **0**：该设备的从设备号（本例中没有用到）。

例如，若在 `/var/log/messages` 中看到的主设备号是 254，则该命令看起来可能是这样：

```
Lkp:~# mknod /dev/par11 c 254 0
```

5. 应用程序代码

此处，创建一个简单的应用程序来打开模块，并在输出引脚 D0~D7 上进行二进制计数。使用命令 `gcc app.c` 编译该代码，可执行输出默认为 `a.out`：

```
-----
app.c
000 //application to use parallel port driver
```

```

#include <fcntl.h>
#include <linux/ioctl.h>
004 #include "parll.h"

main()
{
    int fptr;
    int i,retval,parm =0;

    printf("\nopening driver now");
012    if((fptr = open("/dev/parll",O_WRONLY))<0)
    {
        printf("\nopen failed, returned=%d",fptr);
        exit(1);
    }

018    for(i=0;i<0xff;i++)
    {
020        system("sleep .2");
021        retval=ioctl(fptr,DATA_OUT,parm);
022        retval=ioctl(fptr,GET_STATUS,parm);

024        if(!(retval & 0x80))
            printf("\nBusy signal count=%x",parm);
        if(retval & 0x40)
027            printf("\nAck signal count=%x",parm);
028        // if(retval & 0x20)
        // printf("\nPaper end signal count=%x",parm);
        // if(retval & 0x10)
        // printf("\nSelect signal count=%x",parm);
        // if(retval & 0x08)
033        // printf("\nError signal count=%x",parm);

        parm++;
    }
038    close(fptr);
}

```

第 4 行：应用程序和驱动程序的头文件通常都包含新的 IOCTL 辅助宏来进行类型检查。

第 12 行：打开驱动程序，获得模块的文件句柄。

第 18 行：进入循环。

第 20 行：减缓循环，以便监视 lights/count。

第 21 行：使用文件指针发送 DATA_OUT 命令到该模块，该模块又使用 outb() 将参数的低 8 位数据写入数据端口。

第 22 行：使用 ioctl 函数的 GET_STATUS 命令读取状态字节。这一过程通过 inb() 来实现，并返回读到的字节数。

第 24~27 行：监视感兴趣的特殊数据位。要注意的是，Busy* 是一个低有效的信号，因而当 I/O 完成时读到的这个值为真。

第 28~33 行：改进设计时取消这些注释。

第 38 行：关闭模块。

如果你已经建立了如图 5-5 所示的连接器，当计数器的最高两位数据置位时，产生了 busy 和 ack 信号。应用程序读取这些数据位并进行相应地输出。

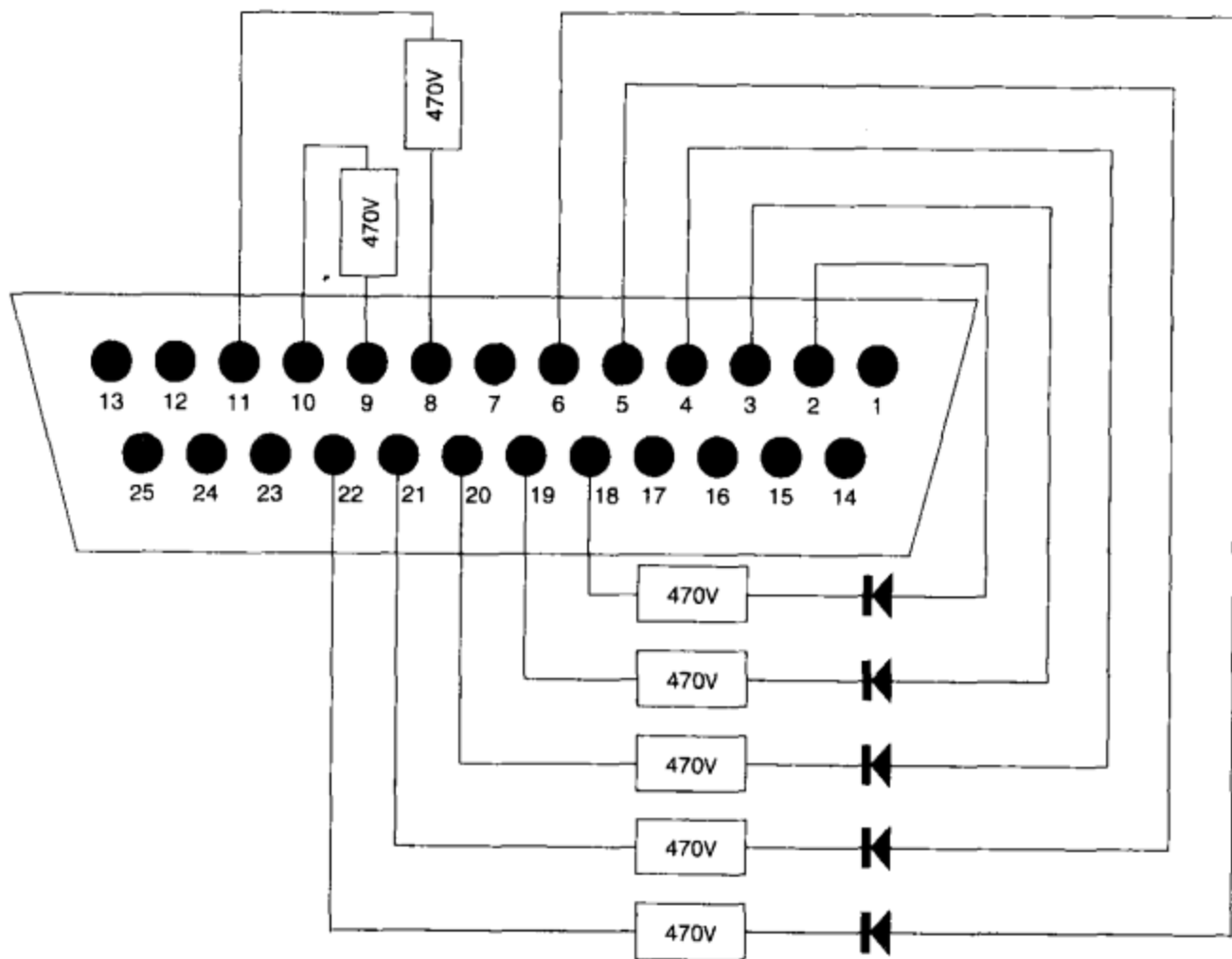


图 5-5 连接器

以上仅简略叙述了字符设备驱动程序的主要内容。了解这些函数后，跟踪工作代码或创建自己的驱动程序都变得简单起来。当给该模块添加中断处理程序时，涉及对函数 `request_irq()` 的调用，同时还要传递所需的 IRQ 号和中断处理程序的名字。这些工作都应该在函数 `init_module()` 中来完成。

建议在该驱动程序中添加如下内容。

□ 使并口模块服务定时器中断轮询输入。

- 怎样将多路 8 位 I/O 数据扩展到 16 位，32 位，64 位？要付出什么代价？
- 从模块的写例程发送字符到串口。
- 使用 ack 信号增加一个中断例程。

5.5 习题

1. 加载一个模块。该模块在文件系统中会变成什么类型的设备文件？
2. 找出已加载的设备文件的主设备号和从设备号。

3. 什么时候采用最后期限 I/O 调度程序比预期 I/O 调度程序更有利一些?
4. 什么时候采用空操作 I/O 调度程序比预期 I/O 调度程序更好一些?
5. 北桥控制器和南桥控制器各有什么特点?
6. 将诸多功能块加入超级输入输出芯片中有何好处?
7. 为什么现在没有将图形和网络通讯功能块加入到超级输入输出芯片中?
8. ext3 之类的日志型文件系统与 ext2 之类的标准文件系统相比,其主要差别是什么?有何优势?
9. 预期 I/O 调度程序的基本理论是什么?这种方法更适合硬盘驱动器和 RAM 盘吗?
10. 块设备和字符设备的主要区别是什么?各举一例。
11. DMA 是什么?为什么说它是一种数据传输的有效方式?
12. 电传打字机最初用在何处?



第 6 章

文件系统

本章内容

- 文件系统的基本概念
- Linux 虚拟文件系统
- 与 VFS 相关的结构
- 页缓存
- VFS 的系统调用和文件系统层

信息处理是围绕信息的存储、检索以及操作而展开的。

在第 3 章中，我们对进程是程序执行的基本单元这一主题进行了讨论，对进程处理信息的方式是把自己的实体存放在其地址空间这一形式进行了考察。然而，进程的地址空间只存在于进程的整个生命周期中，而且它只占系统内存大小的一小部分。文件系统的发展源于对大容量、非易失性信息存放在外存而不是存放在寄存器和内存这样的需求。非易失性信息指始终存在的数据，不管处理信息的进程是否结束，不管操作系统是否停止。

信息存放在外存又引出信息如何表示的问题。信息存储的基本单元是**文件**（file）。**文件系统**（filesystem）或文件管理子系统，是操作系统的组成部分，它处理文件结构、文件操作和文件保护。本章就有关 Linux 文件系统实现的问题展开讨论。

6.1 文件系统的基本概念

首先，我们描述 Linux 文件系统所涵盖的概念。大多数人对这些概念已经相当熟悉，因为在 Linux 的使用及用户空间应用程序编程中都涉及这些概念。如果你已经掌握了文件系统的基本概念，可以直接跳到 6.2 节。

6.1.1 文件和文件名

“文件”这个单词是从现实世界借用的术语。自电子管出现以来，信息就存储在文件中。现实世界的文件是由一张或几张预定大小的纸组成的。这些文件通常存放在橱柜中。

在 Linux 中，文件是一种线性的字节流。这些字节的含义对于操作系统来说并不重要，但是对于用户来说至关重要，就像橱柜对文件的内容毫不关心一样。文件系统提供存储数据的用户接口，透明地操作来自外部设备的物理数据。

Linux 中的文件有许多属性和特征。用户最为熟悉的属性通常是文件的名称。文件的名称常常能表明文件的内容。文件名可能含有**文件扩展名**，扩展名是用句号添加到主文件名之后的附加名。这种扩展名为辨别用户空间应用程序的内容提供了额外的方法。例如，迄今为止，我们看到的所有例子的文件扩展名都是.h 或.c。用户空间的程序，例如编译器和链接器，分别用.h 和.c 表示文件是头文件或源文件。

尽管文件名对用户应用（例如编译器）来说可能很重要，但是操作系统毫不关心文件名，因为它只把文件视为容纳字节的容器，而不考虑文件的内容或用途。

6.1.2 文件类型

Linux 支持多种文件类型，包括普通文件、目录文件、链接文件、设备文件、套接字及管道文件。**普通文件**（regular file）包括二进制文件和 ASCII 文件。ASCII 文件仅仅是一些文本行，用户无需解释程序就能看到和读懂它。某些 ASCII 文件是可执行的，称为**脚本**（script）。这些文件由解释程序来执行。从根本上来讲，shell 也是一个解释程序。可执行的二进制文件是非 ASCII 文件，其内容表面上看起来像是随机数据。这些文件有一个内部格式，内核通过解释这个格式来运行程序。这个格式称为**目标文件格式**（object file format），每个操作系统都解释预定义的目标文件格式。第 9 章将更详细地介绍目标文件格式。

在 Linux 中，文件被组织成多级目录系统，比如图 6-1 所显示的系统。**目录**（directory）包含文件，用来维护文件系统的结构。以下内容会更详细地考察目录和 Linux 的文件结构。

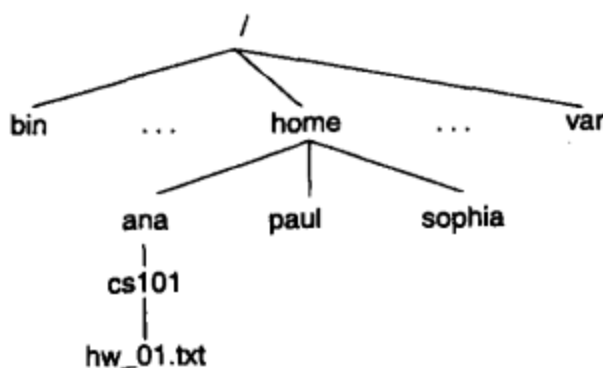


图 6-1 文件系统的层次

链接（link）是指向另一个文件的文件，即文件指针。这些文件仅含有访问另一个文件的必要信息。

设备文件（device file）是 I/O 设备的表示，通过设备文件可以对硬件设备进行访问。程序需要访问 I/O 设备时，使用文件中对应的属性就可以操作文件所代表的设备。有两种主要类型的设备：**块设备**和**字符设备**。块设备（block device）以块为单位传送数据；字符设备（character device）以字符为单位传送数据。第 5 章已介绍了 I/O 设备的详细内容。

套接字文件和**管道文件**是进程间通信的方式。这些文件支持进程间的定向数据流动。我们不讨论这些特殊文件。

6.1.3 文件的附加属性

除了文件名、文件类型和文件数据外，文件还有更多的属性。操作系统会为每个文件添加附

加信息，例如文件的访问权限。在多用户系统（例如 Linux）中，文件保护变得越来越重要。用户被分成三类：

- 用户或文件的所有者；
- 组用户或与文件所有者同组的用户；
- 其他用户，指系统中所有不属于文件组的其余用户。

上述每一类用户对文件分别拥有一组特定的权限。尽管对文件可以进行许多操作，但 Linux 将这些操作归纳为三类并分别对应于三种权限：读、写和执行。因为这三种权限均可应用于三类用户中的每一类，因此，一个文件可能会拥有 9 种不同的权限组合。

文件的其他属性包括文件大小、创建时间戳和最后一次访问的时间戳，所有这些属性都可以通过核心实用程序 `ls` 显示出来。当我们研究文件在内核如何实现时，我们会看到用户看不见的其他许多属性。

6.1.4 目录和路径名

目录是用来维护文件系统层次结构的文件。目录中存放它所含有文件、下级目录以及目录本身的相关信息。在 Linux 中，每个用户都有自己的“主目录”，用户把自己的文件存放在这个目录下，并在这个目录下创建他自己的目录树结构。在图 6-1 中，我们可以看到，目录构成了文件系统的树形结构。

把文件系统组织成树形结构后，只凭文件名就很难找到文件了。我们必须知道文件在树中所处的位置才能找到它。文件的**路径名**（pathname）描述了文件的位置。文件的位置可以相对于树的根目录来描述，这种路径名称为**绝对路径名**（absolute pathname）。绝对路径名从树的根目录（也就是/）开始，目录节点使用目录名后跟“/”来表示，例如 `bin/`。因此，文件的绝对路径名就是由一系列目录节点组成的，沿着这些节点在文件系统的树形结构中一层一层下探，最终就会找到相应的文件。在图 6-1 中，文件 `hw1.txt` 的绝对路径名是 `/home/ana/cs101/hw1.txt`。表示文件的另一个方法是用**相对路径名**（relative pathname）。相对路径名依赖于和文件相关的进程的工作目录。工作目录（working directory），或当前目录（current directory），指与进程执行相关的目录。因此，如果 `/home/ana/` 是进程的工作目录，我们就可以把文件称为 `cs101/hw1.txt`。

在 Linux 中，目录包含那些在操作系统运行期间执行不同任务的文件。例如，共享文件存放在 `/usr` 和 `/opt` 目录下，而非共享的文件存放在 `/etc/` 和 `/boot` 目录下。同样地，非静态文件（文件内容可被系统程序修改的文件）存放在 `/var` 下面的 `vcertain` 目录下。关于文件系统层次标准的更多信息请参见 <http://www.pathname.com/fhs>。

在 Linux 中，每个目录有两个与其相关的入口：`.`（发音为“点”）和`..`（发音为“点点”）。入口`.`表示当前目录，而`..`表示其父目录。对根目录来说，`.`和`..`都表示当前目录。（换句话说，根目录是它自己的父目录。）这种表示法使得用以下方式表示相对路径更为方便了。在前面的例子中，工作目录是 `/home/ana`，我们文件的相对路径名是 `csw101/hw1.txt`。如果要在我们的工作目录中表示 `paul` 目录中的文件 `hw1.txt`，就可以使用 `../paul/cs101/hw1.txt` 这个相对路径名，开头的`..`表示先向上返回一层。

6.1.5 文件操作

文件操作包括系统所允许对文件执行的所有操作。一般来说,对文件的操作包括创建和撤消、打开和关闭、读和写,此外,还能对文件重命名、更改它的属性。文件系统提供系统调用,作为完成这些操作的接口,这些系统调用又被置入封装函数中,用户空间的程序可经由链接库来访问这些封装函数。当我们考察 Linux 文件系统的实现时,会深入探究其中的部分操作。

6.1.6 文件描述符

文件描述符就是一个整型数,系统以此来识别打开的文件。`open()` 系统调用返回一个文件描述符,当进程以后要访问这个文件时,对文件的所有操作都可以使用这个文件描述符。在下一节中,我们会看到在内核术语中文件描述符所代表的含义。

每个进程包含一个文件描述符数组。当我们讨论内核为文件系统提供的数据结构时,我们会看到如何用数组管理文件描述符。依照惯例,数组的第一个元素(文件描述符 0)对应进程的标准输入,第二个元素(文件描述符 1)对应标准输出,而第三个元素(文件描述符 2)对应标准错误输出。这就允许应用程序打开标准输入、标准输出及标准错误文件。图 6-2 说明了属于进程的文件描述符数组。

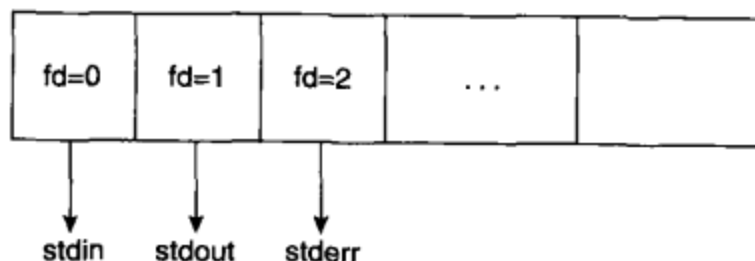


图 6-2 文件描述符数组

文件描述符的分配是以“最小可用索引”的基本原则进行的。因此,如果进程打开多个文件,为它分配的文件描述符就是递增的,如果以前打开的文件在新文件打开之前关闭,就不一定递增。我们可以看到 `open` 系统调用和 `close` 系统调用为了确保这个原则将会如何操作文件描述符。因此,在进程的整个生命期中,进程可能拥有两个不同的文件,但对应相同的文件描述符,这种情况便是先关闭一个文件,然后又开另一个文件。相反,两个不同的文件描述也可能指向同一个文件。

6.1.7 磁盘块、磁盘分区以及实现

为了关注文件系统的实现,我们必须理解关于硬盘的一些基本概念。硬盘利用磁性记录数据。硬盘含有记录数据的多个旋转式盘片。磁头被安装在可在盘片表面来回移动的机械臂上,通过沿着磁盘半径的移动来读写数据,就好像唱片机的针一样。而磁盘片本身像唱片机上面的唱片一样旋转。每张盘片分成多个叫做磁道的同心环。磁道由外向里进行编号。磁道号相同的一组磁道(跨越所有磁盘)叫做柱面。每个磁道(通常)又被分成 512 字节的扇区。柱面、磁道、磁头构成了硬盘驱动器的几何特性。

空磁盘必须在文件系统建立前进行格式化,格式化会创建磁盘的磁道、盘块和分区。分区指

逻辑磁盘，操作系统利用它分配或使用硬盘。我们可以借助于分区把单个磁盘划分为多个部分，使之看起来好像有多个磁盘一样。这使得不同的文件系统可以存在于同一张盘上。每个分区又被分成磁道和块。磁盘中**磁道和块**的创建是由 fdformat^①这样的程序完成的，而逻辑分区的创建由 fdisk 这样的程序完成。磁道和块的创建都是在实际的文件系统创建之前就完成的。

Linux 的文件树可以提供对多个文件系统的访问。这意味着如果你有一个多分区的磁盘，而每个分区都有一个文件系统的话，那么，从一个逻辑命名空间就可能访问到所有这些文件系统。使用 mount 命令可以把每个文件系统加入到 Linux 的主文件系统树中。我们说一个文件系统被安装是指设备文件系统被加入主树，从而可以从主树中访问设备文件系统。文件系统被安装到目录上^②。安装文件系统的目录叫做挂载点 (mount point)。

文件系统实现的主要难点之一在于抉择操作系统对组成文件的字节序列到底如何存放。如前所述，磁盘的分区空间被划分为叫做块的一簇簇空间。块的大小因实现的不同而不同。块的管理方式决定文件的访问速度和碎片的等级^③，并因此而浪费一些空间。例如，如果我们有一个大小为 1 024 字节的块和一个大小为 1 567 字节的文件，显然，这个文件横跨两个块。操作系统通过维护一个叫做索引节点 (inode) 结构中的信息，就可以记录某个文件中块的情况。

6.1.8 性能

文件系统具有提高系统性能的各种方法，其中之一就是通过在内核中维护内部基础结构来快速访问一个给定路径名对应的索引节点。当我们解释文件系统的实现时，将会看到内核是如何实现这种方法的。

页缓存是文件系统提高性能的另一个方法。页缓存是内存中页的集合。它被设计成缓存许多不同类型的页，这些页来源于磁盘文件、内存映射文件或内核能够访问的其他任何页对象。这种缓存机制大大减少了访问磁盘的次数，提高了系统性能。本章将阐述在文件管理的过程中，页缓存是如何与磁盘访问相结合的。

6.2 Linux 虚拟文件系统

文件系统的实现因系统的不同而不同。例如，在 Windows 和 Unix 中，文件与磁盘块如何关联的具体实现就有很大的不同。实际上，微软实现了各种操作系统对应的多种文件系统：DOS 和 Win 3.x 对应的 MS-DOS，Windows 9x 对应的 VFAT，Windows NT 对应的 NTFS。UNIX 操作系统也实现了多种文件系统，例如 SYSV 和 MINIX。特别是，Linux 使用 ext2、ext3 和 ReiserFS 之类的文件系统。

Linux 最具特色的一点就是支持多种文件系统。你不仅能够从它自己的文件系统中 (ext2、ext3、和 ReiserFS) 查看文件，而且能从与其他操作系统相关的文件系统中查看文件。在单独的 Linux 系统中，你能够访问许多不同格式的文件。表 6-1 列出了 Linux 目前支持的文件系统。对

① fdformat 用于软盘的低级格式化 (建立磁道和扇区)。IDE 硬盘和 SCSI 硬盘通常在出厂时就进行了预格式化。

② 用树的说法，你就可以说把子树添加到主树的一个节点上。

③ 第 4 章中介绍了碎片，以及内存中的空洞是如何产生的，相同的碎片问题在硬盘存储中也会遇到。

用户来说,从一种文件系统跨越到另一种文件系统没有任何差异;他能随心所欲地把支持的文件系统安装到最初的那个树命名空间。

表6-1 Linux支持的部分文件系统

文件系统名	说 明
ext2	第二扩展文件系统
ext3	ext3日志文件系统
Reiserfs	日志文件系统
JFS	IBM的日志文件系统
XFS	SGI Irix的高性能日志文件系统
MINIX	Linux最初的文件系统, minix操作系统的文件系统
ISO9660	CD-ROM文件系统
JOLIET	微软的CRDOM文件系统扩展
UDF	可选的CROM、DVD文件系统
MSDOS	微软的磁盘操作系统
VFAT	Windows 95的虚拟文件分配表
NTFS	Windows NT、2000、XP、2003的文件系统
ADFS	Acorn磁盘文件系统
HFS	Apple的Macintosh文件系统
BEFS	BeOs文件系统
FreeVxfs	Veritas Vxfs支持
HPFS	OS/2支持
SysVfs	系统V的文件系统支持
NFS	网络文件系统支持
AFS	Andrew文件系统 (也是网络文件系统)
UFS	BSD的文件系统支持
NCP	NetWare文件系统
SMB	Samba

Linux 不但支持多种磁盘文件系统,还支持网络安装的文件系统,以及用于处理某些专门的事情而不是管理磁盘空间的特殊系统。例如, `procfs` 就是一个伪文件系统。这个虚拟的文件系统提供关于系统各个方面的信息。`procfs` 文件系统不占据硬盘空间,它的文件在访问时动态创建。另一个这样的文件系统是 `devfs`^①,它为设备驱动程序提供接口。

通过在用户空间和实际文件系统之间引入一个抽象的中间层, Linux 完成对实际文件系统细节的“伪装”。这个层称为 VFS (virtual filesystem, 虚拟文件系统)。它把文件系统所特有的结构和来自内核其余部分的功能隔离开来。VFS 管理与文件系统相关的系统调用,并把它们转化成适当文件系统类型的函数。图 6-3 概括了文件系统管理的结构。

① 在 Linux 2.6 中,尽管仍然少量地支持 `devfs`,但是它被 `udev` 取代了。有关 `udev` 的更多信息,参见 <http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev-FAQ>。

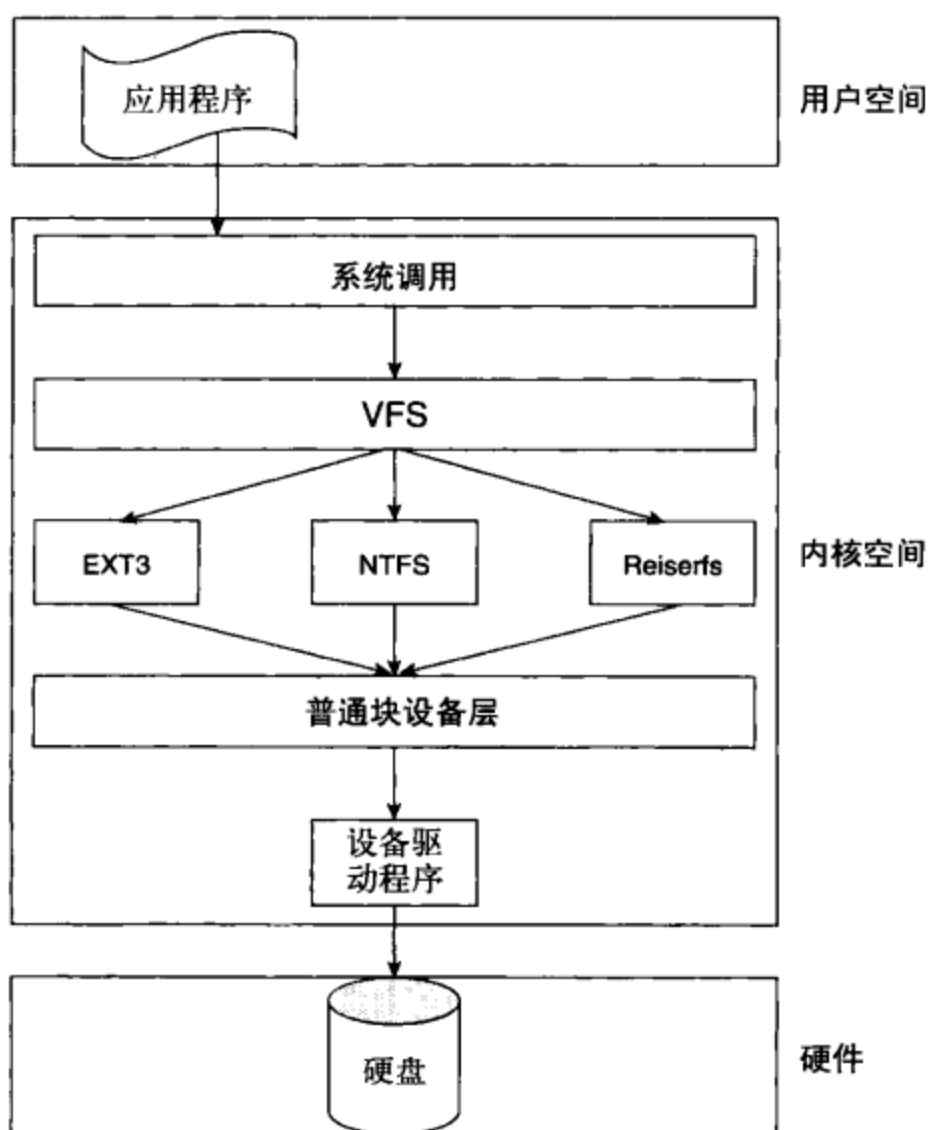


图 6-3 Linux 的虚拟文件系统

用户应用程序通过系统调用访问通用的 VFS。每个被支持的文件系统必须实现一组函数，这组函数完成 VFS 所支持的操作（例如，打开、读、写和关闭文件）。VFS 记录它所支持的文件系统，并给出执行每个操作的函数。从第 5 章可以知道，在文件系统和实际的设备驱动程序之间有一个普通的块设备层。这提供了一个抽象层，使得文件系统特有代码的实现与它最终访问的具体设备无关。

6.2.1 VFS 的数据结构

VFS 利用数据结构来表示文件系统。

该数据结构如下所示。

- 超级块结构。存放已安装的文件系统的相关信息。
- 索引节点结构。存放有关文件的信息。
- 文件结构。存放被进程打开的文件的相关信息。
- 目录项结构。存放有关路径名及路径名所指向的文件的信息。

除了这些结构外，VFS 还使用一些附加结构，例如 `vfsmount` 和 `nameidata`，它们分别存放安装信息和路径的查找信息。尽管我们不会单独地介绍它们，但是要看看这两个结构如何与刚才所描述的主要结构相联系。

在 VFS 中有一些与操作相关的结构，这些结构所表示的对象就是操作所施加的实体。在每个对象的操作表中都定义有这些操作。操作表实际上就是函数指针的列表。描述它们的时候，我们为每个对象定义操作表。现在，我们具体分析一下这些结构体。（注意，为了简短、清晰起见，我们没有考虑任何加锁机制）。

1. 超级块结构

当文件系统被安装时，有关它的所有信息均被存放在 `super_block` 结构体中。每个安装好的文件系统都有一个超级块结构。我们先说明这个结构的定义，然后，对比较重要的一些字段进行解释：

```
-----
include/linux/fs.h
666 struct super_block {
667     struct list_head    s_list;
668     dev_t                s_dev;
669     unsigned long        s_blocksize;
670     unsigned long        s_old_blocksize;
671     unsigned char        s_blocksize_bits;
672     unsigned char        s_dirt;
673     unsigned long long    s_maxbytes;
674     struct file_system_type *s_type;
675     struct super_operations *s_op;
676     struct dquot_operations *dq_op;
677     struct quotactl_ops    *s_qcop;
678     struct export_operations *s_export_op;
679     unsigned long        s_flags;
680     unsigned long        s_magic;
681     struct dentry         *s_root;
682     struct rw_semaphore    s_umount;
683     struct semaphore      s_lock;
684     int                   s_count;
685     int                   s_syncing;
686     int                   s_need_sync_fs;
687     atomic_t              s_active;
688     void                  *s_security;
689
690     struct list_head      s_dirty;
691     struct list_head      s_io;
692     struct hlist_head     s_anon;
693     struct list_head      s_files;
694
695     struct block_device    *s_bdev;
696     struct list_head      s_instances;
697     struct quota_info      s_dquot;
698
699     char                   s_id[32];
700
701     struct kobject         kobj;
702     void                  *s_fs_info;
703     ...
708     struct semaphore      s_vfs_rename_sem;
709 };
-----
```

第 667 行：s_list 字段是 list_head 类型^①的，它是指向双向循环链表中前一个元素和下

① 第 2 章详细地描述了 list_head 数据类型。

一个元素的指针，这个超级块被嵌入双向循环链表中。如同 Linux 内核中的其他结构体一样，`super_block` 结构体也是用一个双向循环链表来维护的。`list_head` 数据类型包含指向其他两个 `list_head` 的指针：前一个超级块对象的 `list_head` 和下一个超级块对象的 `list_head`。（全局变量 `super_blocks` (`fs/super.c`) 指向链表中的第一个元素。）

第 672 行：在基于磁盘的文件系统中，用最初存放在磁盘特定扇区中的信息（从磁盘装入到内存的超级块结构中）来填充超级块结构，因为 VFS 允许对超级块结构中的字段进行修改，所以，超级块结构中的信息可能发现自己与磁盘上的数据不同步。这个字段标识超级块结构已被修改，于是需要和磁盘进行同步。

第 673 行：这个无符号长整型的字段定义文件系统所允许的最大文件大小。

第 674 行：超级块结构体包含通用文件系统的信息。然而，它必须涉及具体文件系统（例如，MSDOS、ext2、MINIX 和 NFS）的信息。结构体 `file_system_type` 存放具体文件系统的信息，内核中配置的每种类型的文件系统都对应一个这种结构体。这个字段指向某个具体文件系统的结构体，VFS 利用它管理从一般请求到具体文件系统操作的交互过程。

图 6-4 给出了 `file_system_type` 和超级块之间的关系。该图说明了 `superblock->s_type` 字段如何指向 `file_systems` 链表中合适的 `file_system_type` 结构体（6.2.2 节说明了 `file_systems` 链表究竟是什么）。

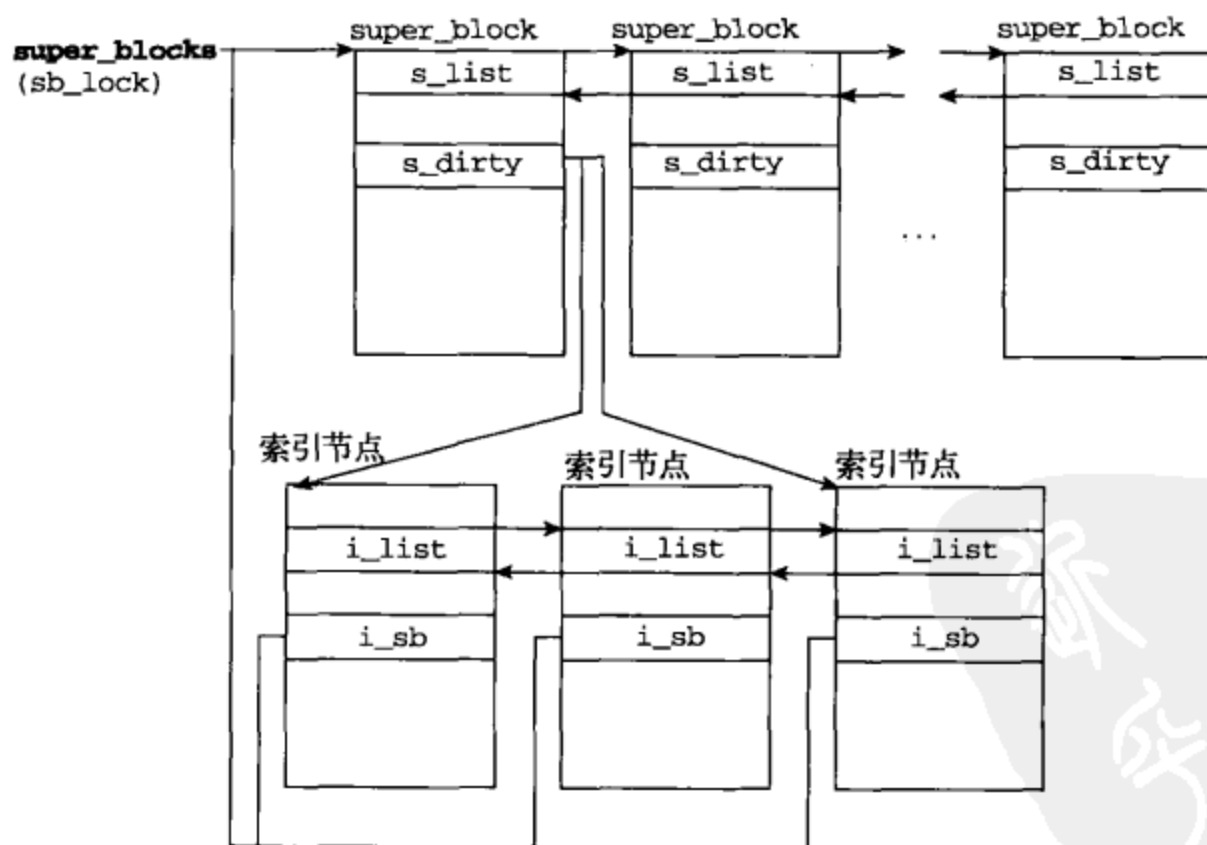


图 6-4 超级块与 `file_system_type` 之间的关系

第 675 行：这个字段是 `super_operation` 结构体类型的指针。这个数据类型存放超级块的操作表。`super_operations` 结构体本身存放一些函数指针，用特定文件系统的超级块操作来初始化这些函数指针。下一节将更详细地解释 `super_operations`。

第 681 行：这个字段是指向目录项结构体的指针。目录项结构体存放文件的路径名。`s_root` 目录项对象的特殊之处在于，它与安装目录相关，属于安装目录的超级块。

第 690 行: `s_dirty` 字段 (不要与 `s_dirt` 混淆) 是一个 `list_head` 结构体, 它指向该文件系统所包含的脏索引节点链表的第一个元素和最后一个元素。

第 693 行: `s_files` 字段是一个 `list_head` 结构体, 它指向文件结构体 (分配给超级块并正在被超级块使用) 链表的第一个元素。在“文件结构”部分, 你会看到它是文件结构所在的三个链表中的一个。

第 696 行: `s_instances` 字段是一个 `list_head` 结构体, 它指向超级块链表中相邻的超级块元素, 该超级块链表中的超级块都是同一种文件系统类型。 `file_system_type` 结构体的 `fs_supers` 字段指向 `s_instances` 链表的表头。

第 702 行: 这个 `void *` 数据类型指向超级块的附加信息, 这些附加信息是具体文件系统所特有的 (例如, `ext3_sb_info`)。对于无法抽象出虚拟文件系统超级块概念的具体文件系统来说, 它就像包罗万象的容器, 存放磁盘上超级块中的任何数据。

2. 超级块的操作

超级块的 `s_op` 字段指向文件系统的超级块可以执行的操作表。每个文件系统都有各自的列表, 因为它直接对文件系统的实现进行操作。操作表存放在一个 `super_operations` 类型的结构体中:

```
-----
include/linux/fs.h
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);

    void (*read_inode) (struct inode *);

    void (*dirty_inode) (struct inode *);
    void (*write_inode) (struct inode *, int);
    void (*put_inode) (struct inode *);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    void (*write_super_lockfs) (struct super_block *);
    void (*unlockfs) (struct super_block *);
    int (*statfs) (struct super_block *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);

    int (*show_options)(struct seq_file *, struct vfsmount *);
};
-----
```

当对文件系统的超级块初始化时, `s_op` 字段被设置成指向合适的操作表。“从通用文件系统层到特定文件系统层”部分将说明在 `ext2` 文件系统中如何实现这个操作表。表 6-2 给出了超级块

操作的列表。其中某些函数是可选的，只需要用所支持文件系统的操作表子集来填充。对于不支持的特定可选函数，就把操作结构体中的对应字段设置为 NULL。

表6-2 超级块的操作

超级块操作名	说 明
alloc_inode	这是Linux 2.6内核中新增的。它分配并初始化超级块中的vfs索引节点。具体的初始化留给特定的文件系统去完成。在索引节点缓冲上调用kmem_cache_create()或kmem_cache_alloc()（参见第4章）完成真正的分配
destroy_inode	这是Linux 2.6内核中新增的。它回收与超级块相关的具体索引节点。调用kmem_cache_free()完成回收
read_inode	读取inode->i_ino字段所指向的索引节点。用磁盘上的数据更新索引节点中的字段。其中inode->i_op字段尤其重要
dirty_inode	把索引节点放入超级块的脏索引节点链表中。利用superblock->s_dirty字段来引用这个双向循环链表的头和尾。图6-5说明了超级块的脏索引节点链表
write_inode	把索引节点信息写到磁盘上
put_inode	从索引节点缓冲释放索引节点。调用iput()来进行释放
drop_inode	当最后一次访问完索引节点时调用
delete_inode	从磁盘删除索引节点。用于那些不再需要的索引节点。调用generic_delete_inode()完成删除
put_super	释放超级块（例如，当卸载文件系统时）
write_super	把超级块的信息写到磁盘上
sync_fs	目前只能被ext3、Resiserfs、XFS和JFS使用，这个函数把脏的超级块结构体数据写到磁盘上
write_super_lockfs	由ext3、JFS、Resiserfs和XFS使用，这个函数先禁止对文件系统做修改。然后更新磁盘的超级块
unlockfs	解除由write_super_lockfs()加的锁
stat_fs	调用它获得文件系统的统计信息
remount_fs	重新安装文件系统时调用它，用来更新任一安装选项
clear_inode	释放索引节点以及与该索引节点相关的所有页面
umount_begin	在安装操作必须被打断时调用
show_options	用于从已安装的文件系统获得文件系统的信息

我们完成了对超级块结构及其操作的介绍，接下来开始详细地探究索引节点的结构。

3. 索引节点结构

我们说过，索引节点是记录文件信息的结构，例如指向包含文件所有数据的数据块的指针。回想一下，目录、设备以及管道（作为例子）在内核中也被看做文件，因此，索引节点也能表示目录、设备和管道。索引节点对象在文件的整个生命周期中都存在，而且包含在磁盘上维护的数据。

为了便于引用，索引节点用链表来存放。其中一种链表是散列表，它能够减少查找特定索引节点的时间。索引节点也可以在三种双向链表中的一个里面找到自己。表 6-3 给出了这三种链表的类型。图 6-5 显示了超级块结构和它的脏索引节点链表之间的关系。

表6-3 索引节点链表

链 表	i_count	是 否 脏	引用指针
有效未使用的链表	i_count = 0	不脏	inode_unused (全局的)
有效在使用的链表	i_count > 0	不脏	inode_in_use (全局的)
脏索引节点的链表	i_count > 0	脏的	超级块的s_dirty字段

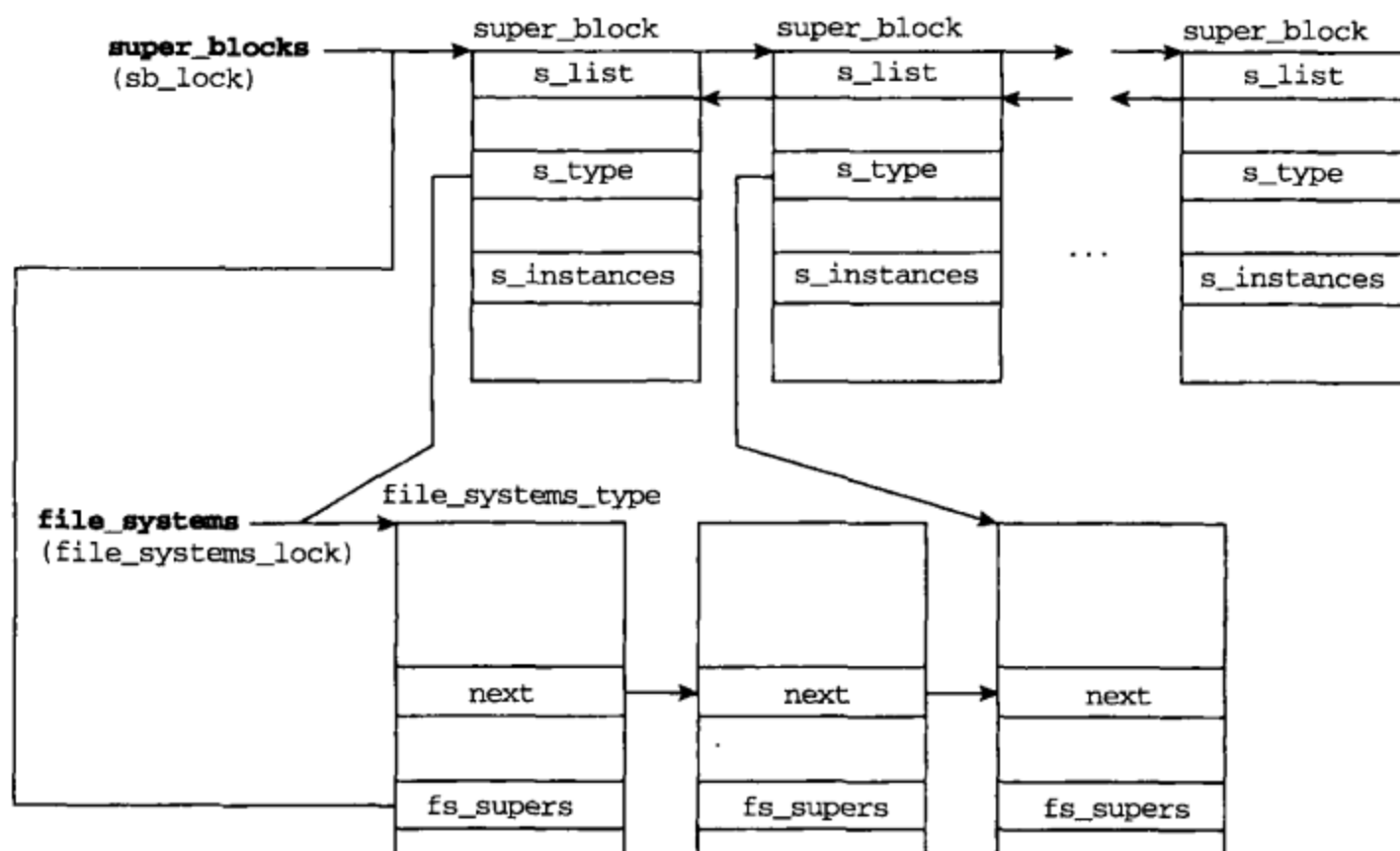


图 6-5 超级块与索引节点之间的关系

索引节点的结构体非常庞大，含有很多字段。下面是对它一小部分字段的描述。

```

-----
include/linux/fs.h
368 struct inode {
369     struct hlist_node  i_hash;
370     struct list_head   i_list;
371     struct list_head   i_dentry;
372     unsigned long      i_ino;
373     atomic_t           i_count;
...
390     struct inode_operations *i_op;
...
392     struct super_block  *i_sb;
...
407     unsigned long      i_state;
...
421 };
-----

```


第 369 行: `i_hash` 字段是 `hlist_node` 类型的^①。它包含一个指向散列表的指针, 散列表用来加速索引节点的查找。索引节点的散列表由全局变量 `inode_hashtable` 来引用。

第 370 行: 这个字段链接索引节点链表中相邻的结构体。索引节点可以在三个链表中的某一个上找到自己。

第 371 行: 这个字段指向文件对应的目录项结构体的链表。目录项结构体中包含该文件所在的路径名, 而文件由索引节点表示。如果文件有多个别名, 则它可能拥有多个目录项结构体。

第 372 行: 这个字段存放唯一的索引节点号。当在一个特定的超级块中分配索引节点时, 这个号是根据前一个已分配的索引节点 ID 自动增加的值。当调用超级块操作 `read_inode()` 时, 这个字段所表示的索引节点被从磁盘读入内存。

第 373 行: `i_count` 字段是一个计数器, 每使用一次索引节点就增加 1。值 0 表示索引节点未被使用, 正数表示索引节点正在使用。

第 392 行: 这个字段存放的指针指向文件所在文件系统的超级块。图 6-5 说明了超级块的脏索引节点链表中的所有索引节点如何让它们的 `i_sb` 字段指向相同的超级块。

第 407 行: 这个字段对应索引节点的状态标志。表 6-4 列出了可能的状态值。

表6-4 索引节点的状态

索引节点的状态标志	说 明
<code>I_DIRTY_SYNC</code>	参见 <code>I_DIRTY</code> 的描述
<code>I_DIRTY_DATASYNC</code>	参见 <code>I_DIRTY</code> 的描述
<code>I_DIRTY_PAGES</code>	参见 <code>I_DIRTY</code> 的描述
<code>I_DIRTY</code>	这个宏关联三个 <code>I_DIRTY_*</code> 标志中的任何一个。它使得能够快速检查这些标志。 <code>I_DIRTY*</code> 标志表示索引节点的内容已被修改, 需要进行同步
<code>I_LOCK</code>	对索引节点加锁时设置, 解锁时清除。第一次创建索引节点以及涉及 I/O 传送时, 索引节点被加锁
<code>I_FREEING</code>	索引节点被删除时设置该标志。因为它将要被删除, 这个标志用于把索引节点标记为不可使用, 这样就没有人能够再引用它了
<code>I_CLEAR</code>	表示索引节点不再有用
<code>I_NEW</code>	索引节点创建时设置。第一次解锁新索引节点时清除此标志

设置了 `I_LOCK` 或 `I_DIRTY` 标志的索引节点可以在 `inode_in_use` 链表中找到自己。如果这两个标志中的一个没有被设置, 索引节点就会被添加到 `inode_unused` 链表中。

4. 目录项结构

目录项结构表示目录的中一项, VFS 用它记录基于目录命名、目录组织以及文件逻辑布局之间的关系。每个目录项对象对应路径名中的一个分量, 并且关联与其相关的其他结构和信息。例如, 在路径 `(/home/lkp)/Chapter06.txt` 中, 包含了 `/`、`home`、`lkp` 这些目录项和 `Chapter06.txt`。每个目录项都涉及分量的索引节点、超级块和相关信息。图 6-6 说明了超级块、索引节点及目录项结构体之间的关系。

^① `hlist_node` 是一个用于双向链表的链表指针类型, 与 `list_head` 非常类似。二者之间的差别是链表头 (`hlist_node` 类型) 只包含一个指向第一个元素的指针, 而不是两个指针 (第二个指向链表尾)。这减少了散列表的开销。

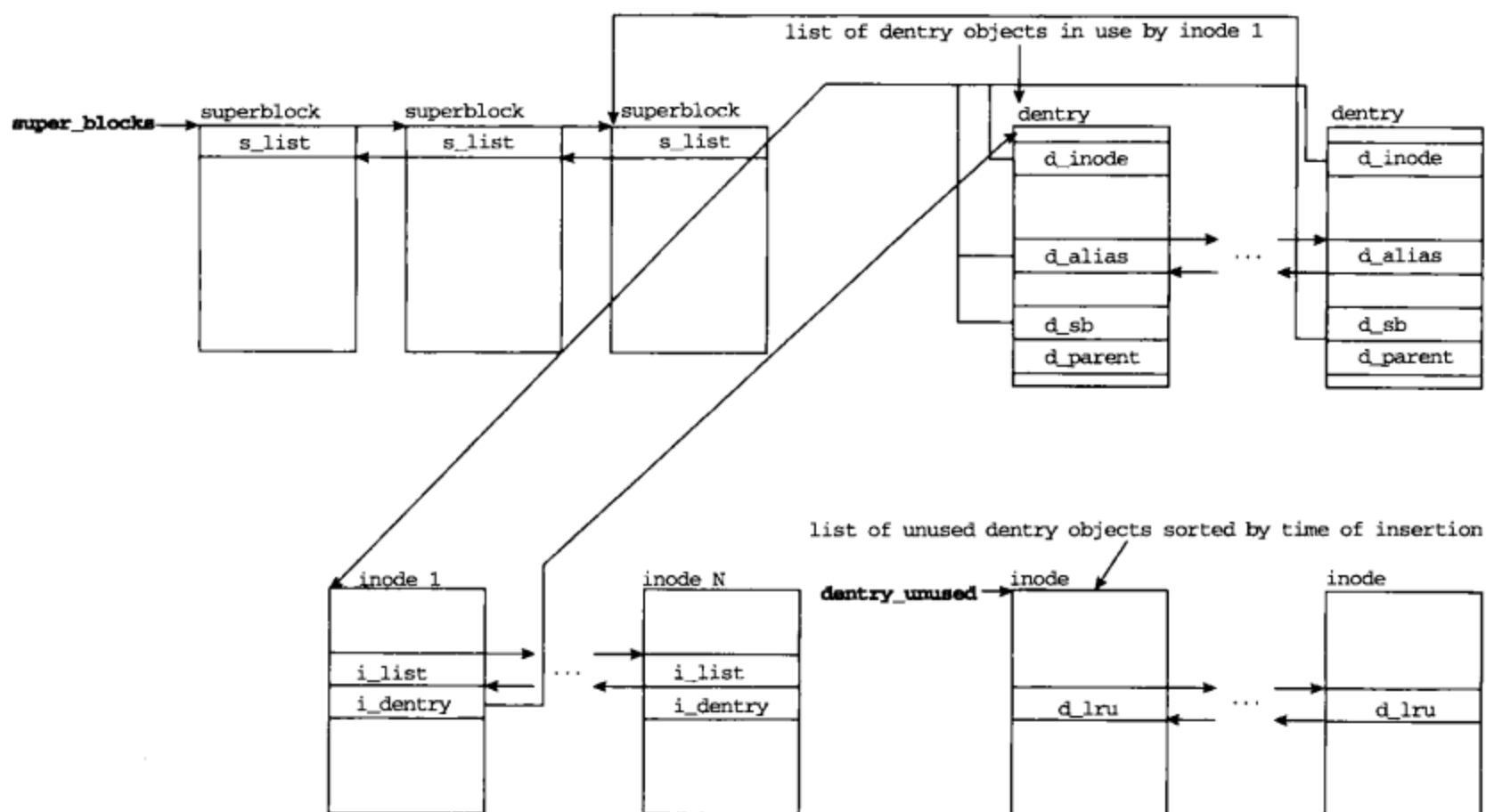


图 6-6 超级块、目录项及索引节点之间的关系

现在来看看 dentry 结构体的部分字段：

```

-----
include/linux/dcache.h
81 struct dentry {
...
85 struct inode * d_inode;
86 struct list_head d_lru;
87 struct list_head d_child; /* child of parent list */
88 struct list_head d_subdirs; /* our children */
89 struct list_head d_alias;
90 unsigned long d_time; /* used by d_revalidate */
91 struct dentry_operations *d_op;
92 struct super_block * d_sb;
...
100 struct dentry * d_parent;
...
105 } ____cacheline_aligned;
-----

```

第 85 行：d_inode 字段指向与目录项关联的文件所对应的索引节点。如果目录项对应的路径名分量没有相关索引节点，这个字段的值就是 NULL。

第 86~88 行：这里是指向目录项链表中相邻元素的指针。目录项对象能够在表 6-5 所示的其中一种链表上找到自己。

表6-5 目录项链表

链表名	链表指针	说明
已使用的目录项	d_alias	索引节点相同的所有目录项形成一个链表，i_dentry 字段指向该链表的表头
未使用的目录项	d_lru	这些目录项不再使用，但是继续保存，以防路径名中再次使用这个分量

第 91 行: `d_op` 字段指向目录项的操作表。

第 92 行: 这是一个指向超级块的指针, 该超级块就是与目录项所表示的分量相关的超级块。参见图 6-6, 可以了解如何把目录项与超级块结构体关联起来。

第 100 行: 这个字段存放指向父目录项或者路径名中父分量对应的目录项的指针。例如, 在路径名 `/home/paul` 中, 目录项 `paul` 的 `d_parent` 字段指向目录项 `home`, 而 `home` 目录项的 `d_parent` 指向目录项 `/`。

5. 文件结构

VFS 使用的另一个结构是文件结构。当进程对文件进行操作时, 文件结构是 VFS 用于存放进程和文件关联信息的数据类型。与其他结构不同, 文件结构并不存放原始的磁盘数据; 文件结构体在调用 `open()` 系统调用时动态创建, 在调用 `close()` 系统调用时被销毁。回想第 3 章, 在进程的整个生命期中, 由进程所打开的表示文件的文件结构体是通过进程描述符 (`task_struct`) 引用的。图 6-7 说明文件结构体如何与 VFS 的其他 VFS 结构体相关联。 `task_struct` 指向文件描述符表, 该表中存放指向由进程打开的所有文件描述符的指针。回忆一下, 描述符表中的前三项分别对应 `stdin`、`stdout` 和 `stderr` 的文件描述符。

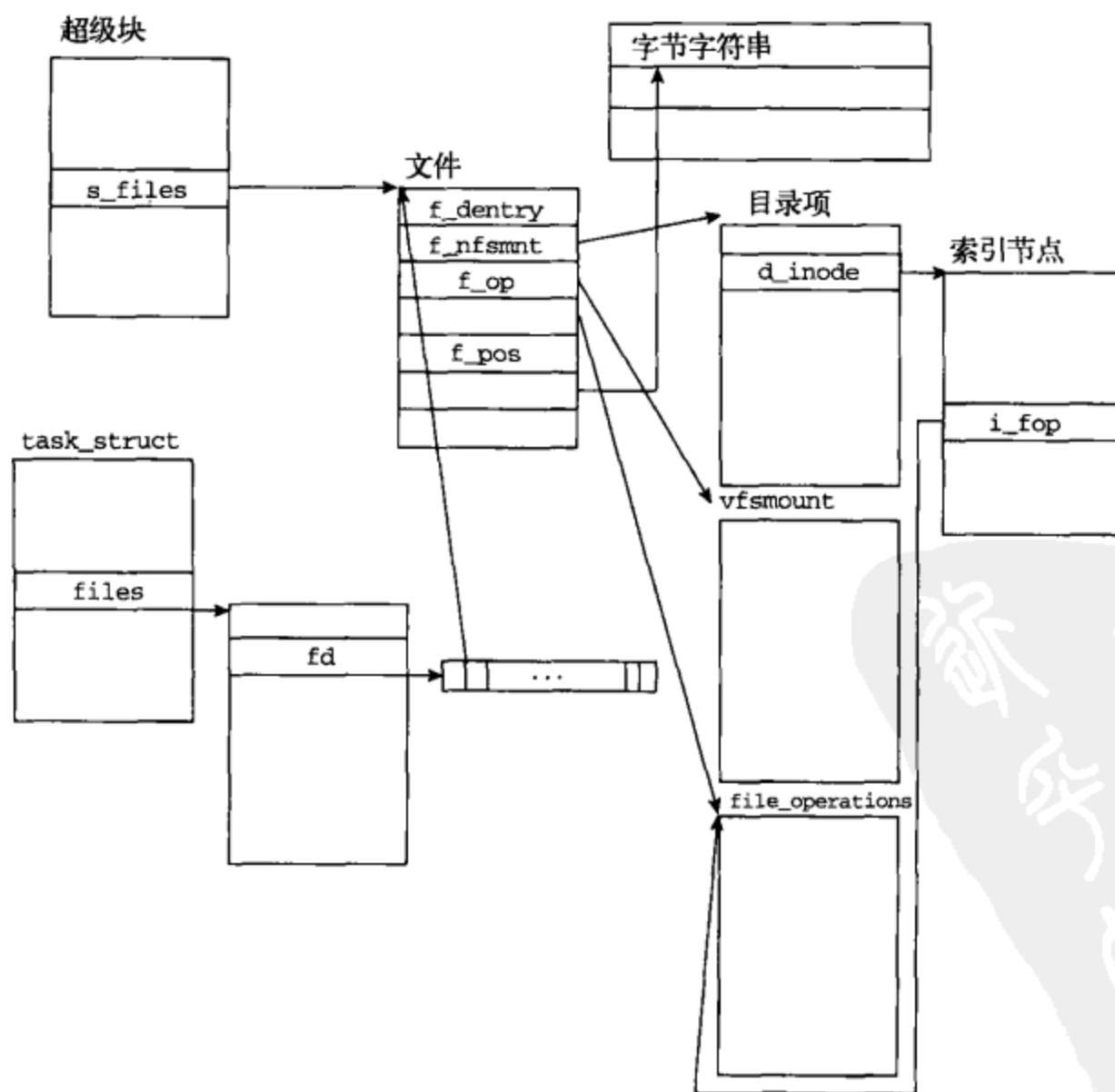


图 6-7 文件对象

内核用双向循环链表来存放文件结构体。根据文件的使用和分配情况, 文件结构体被嵌入三

个链表中的一个。表 6-6 描述了这 3 个链表。

表6-6 文件链表

名 字	指向链表头的指针	说 明
空闲文件对象链表	全局变量 <code>free_list</code>	由所有可用的文件对象组成的双向链表。链表的长度不小于 <code>NR_RESERVED_FILES</code>
正在使用但没有分配的文件对象链表	全局变量 <code>anon_list</code>	由正在被使用但是还没有分配给超级块的所有文件对象组成的双向链表
超级块的文件对象链表	超级块的 <code>s_files</code> 字段	一个双向链表，由包含与超级块相关的文件的所有文件对象组成

内核利用 `get_empty_filp()` 创建文件结构体。该例程返回指向文件结构体的一个指针，如果没有空闲结构体，或是系统已用完了内存就返回 `NULL`。

现在来看看文件结构体中一些比较重要的字段：

```
-----
include/linux/fs.h
506 struct file {
507     struct list_head    f_list;
508     struct dentry       *f_dentry;
509     struct vfsmount     *f_vfsmnt;
510     struct file_operations *f_op;
511     atomic_t            f_count;
512     unsigned int        f_flags;
513     mode_t              f_mode;
514     loff_t              f_pos;
515     struct fown_struct   f_owner;
516     unsigned int        f_uid, f_gid;
517     struct file_ra_state f_ra;
...
527     struct address_space *f_mapping;
...
529 };
-----
```

第 507 行：`list_head` 类型的 `f_list` 字段存放着指向链表中相邻文件结构体的指针。

第 508 行：这是指向与文件相关的目录项结构体的指针。

第 509 行：这是指向 `vfsmount` 结构体的指针，该结构体与文件所在的已安装文件系统相关。所有已安装的文件系统都有一个用于存放安装的相关信息的 `vfsmount` 结构体。图 6-8 给出了与 `vfsmount` 结构体相关的数据结构。

第 510 行：这是一个指向 `file_operations` 结构体的指针，其中存放的文件操作表用于对文件进行操作。（索引节点的字段 `i_fop` 也指向这个结构体。）图 6-7 说明了这种关系。

第 511 行：多个进程可以并发地访问一个文件。当文件结构体未被使用（因此是可用的）时，`f_count` 字段被设置为 0。当文件结构体与一个文件相关联时，`f_count` 被设置为 1，之后，处理文件的进程每增加一个，`f_count` 就增加 1。因此，如果 4 个不同的进程正在访问同一个文件对象，该文件对象正在使用并且表示一个文件，则文件对象的 `f_count` 字段存放的值就是 5。

第 512 行：`f_flags` 字段包含经由 `open()` 系统调用传入的标志。6.5.1 节将更详细地介绍这个

字段。

第 514 行: `f_pos` 字段存放文件的偏移。这实际上就是文件操作表中某些方法用来指示文件当前位置的读/写指针。

第 516 行: 我们必须知道进程的所有者是谁, 以便操作文件时能够确定文件的访问权限。这些字段分别对应启动进程并打开了文件的用户所拥有的 `uid` 和 `gid`。

第 517 行: 文件可以从页缓存中读取页面, 前面说过, 页缓存是内存中页的集合。预读优化包括预读文件中临近的页面, 在这些页面被请求读入内存之前预先读取它们, 可以减少费时的磁盘访问的次数。`f_ra` 字段存放一个 `file_ra_state` 类型的结构体, 它包含了与文件预读状态相关的所有信息。

第 527 行: 这个字段指向 `address_space` 结构体——对应于该文件所使用的页缓存机制。这在 6.4 节中会更详细地讨论。

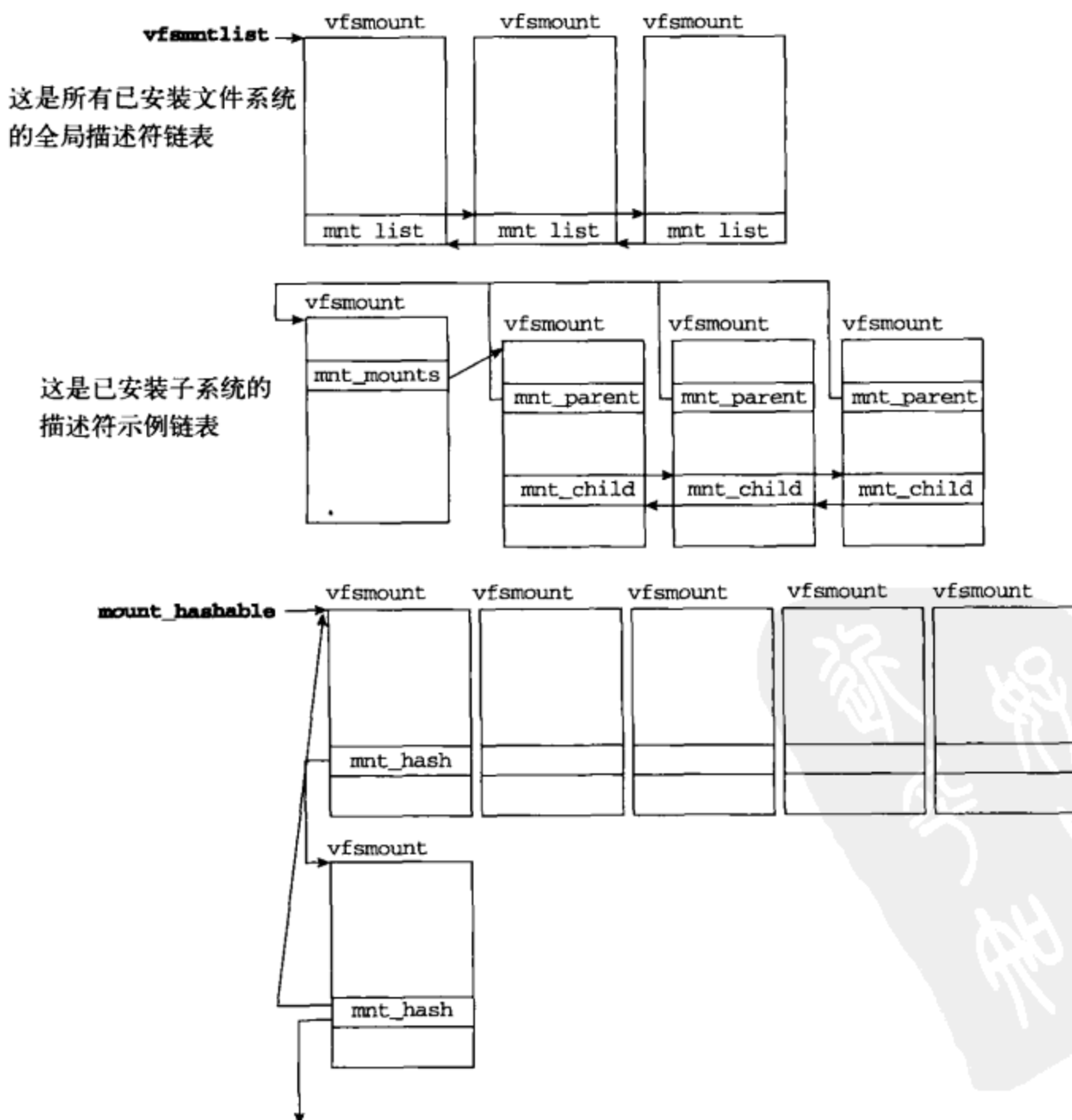


图 6-8 vfmount 对象

6.2.2 全局链表和局部链表的引用

Linux 内核使用全局变量来存放前面提到的指向结构体链表的指针。所有结构体都用双向链表存放。内核存放指向链表头的指针，把它作为链表的访问点。这些结构体都包含 `list_head` 类型的字段^①，用来指向链表中前一个元素和下一个元素。表 6-7 概括了内核存放的全局变量以及这些变量指向的链表的类型。

表6-7 与VFS相关的全局变量

全局变量	结构类型
<code>super_blocks</code>	<code>super_block</code>
<code>file_systems</code>	<code>file_system_type</code>
<code>dentry_unused</code>	<code>dentry</code>
<code>vfsmntlist</code>	<code>vfsmount</code>
<code>inode_in_use</code>	<code>inode</code>
<code>inode_unused</code>	<code>inode</code>

`super_block`、`file_system_type`、`dentry` 和 `vfsmount` 结构体都存放在它们自己的链表中。而索引节点却可能是存放在全局的 `inode_in_use` 上或 `inode_unused` 上，或者是存放在它们所对应的超级块的局部链表上。图 6-9 说明了这些结构体是如何相互关联的。

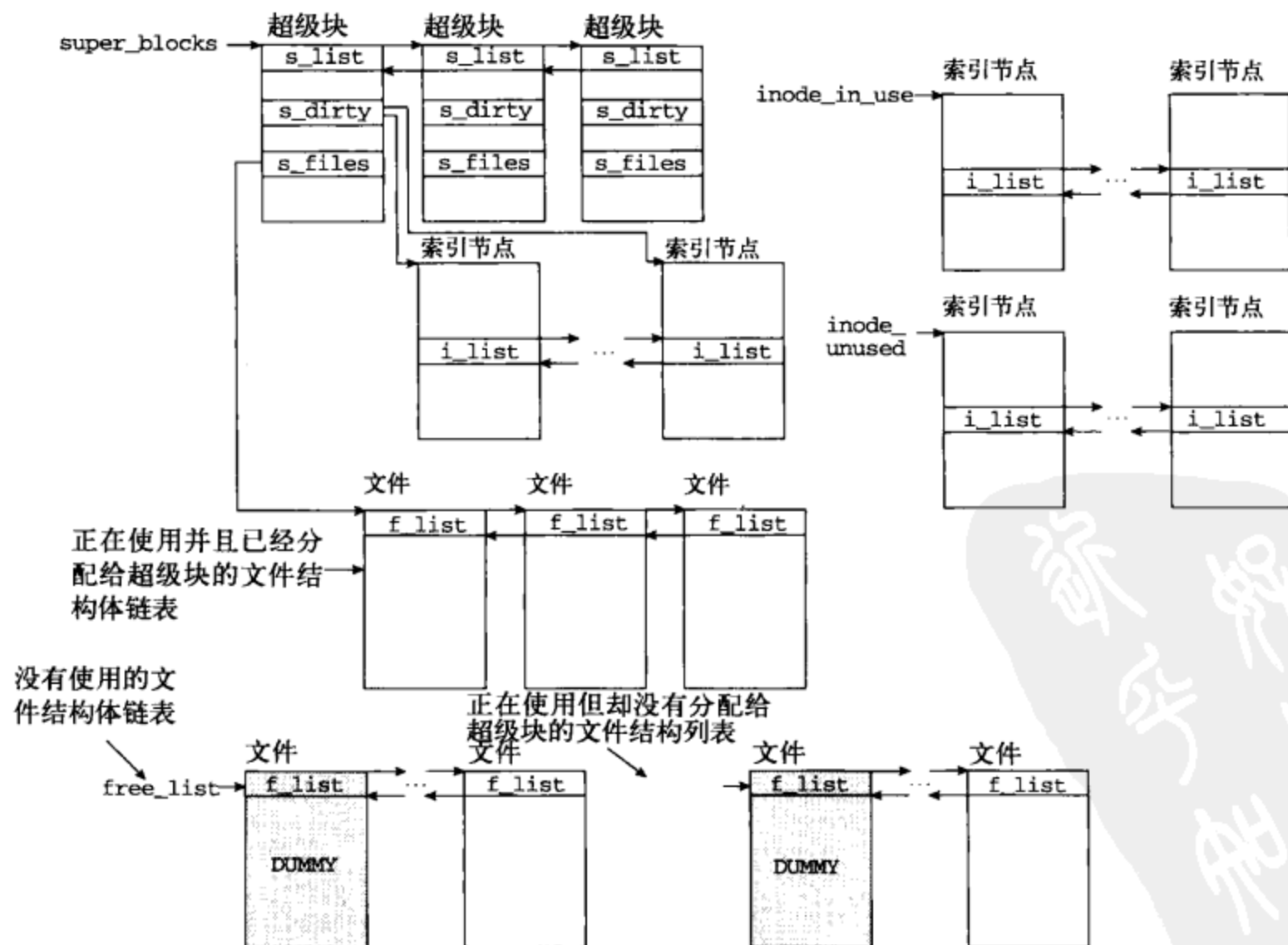


图 6-9 与 VFS 相关的全局变量

① 与我们在“索引节点结构”部分看到的一样，索引节点结构体有一个称为 `hlist_node` 的变体。

`super_blocks` 变量指向超级块链表的头，通过 `s_list` 字段指向链表中前一个元素和下一个元素。超级块结构的 `s_dirty` 字段又依次指向它所拥有的索引节点，这些节点必须与磁盘同步。不在局部的超级块链表上的索引节点就在 `inode_in_use` 链表或 `inode_unused` 链表上。所有索引节点通过 `i_list` 指向索引节点链表中前一个元素和下一个元素。

超级块也通过 `s_files` 指向一个链表的表头，这个链表包含着分配给本超级块的文件结构体。没有分配的文件结构体放到 `anon_list` 链表或 `free_list` 链表中。这两个链表都有一个虚的文件结构体作为链表的表头。所有文件结构体都通过 `f_list` 字段指向其链表中的前一个元素和下一个元素。

参见图 6-6，看看索引节点如何利用 `i_dentry` 指向目录项结构链表。

6.3 与 VFS 相关的结构

除了上述 4 个主要的 VFS 结构之外，还有其他几个结构与 VFS 相互关联：`fs_struct`、`files_struct`、`namespace` 和 `fd_set`。其中，`fs_struct`、`files_struct` 和 `namespace` 结构都是与进程相关的对象，它们包含了与文件相关的数据。图 6-10 说明进程描述符如何与文件相关的结构体相关联。现在，我们来看看这些附加的结构。

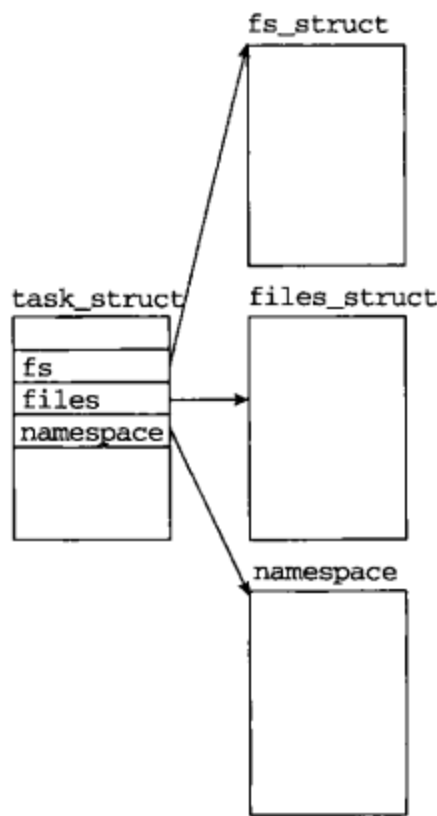


图 6-10 与进程相关的对象

6.3.1 `fs_struct` 结构

在 Linux 中，多个进程能够引用同一个文件。因此，Linux 的 VFS 必须存放有关进程和文件如何交互的信息。例如，在文件操作的权限方面，由某个用户启动的进程和由另一个用户启动的进程是截然不同的。`fs_struct` 结构体中存放将特定进程和文件关联起来的所有信息。在我们考察 `files_struct` 之前必须先考察 `fs_struct` 结构体，因为 `files_struct` 使用了 `fs_struct`

数据类型。

`fs_struct` 可能被多个进程描述符引用,因此,代表文件的 `fs_struct` 由多个 `task_struct` 描述符引用并不稀奇。

```
-----
include/linux/fs_struct.h
7  struct fs_struct {
8      atomic_t count;
9      rwlock_t lock;
10     int umask;
11     struct dentry * root, * pwd, * altpwd;
12     struct vfsmount * rootmnt, * pwdmnt, * altpwrmnt;
13 };
-----
```

1. `count`

`count` 字段存放引用具体 `fs_struct` 的进程描述符的数目。

2. `umask`

`umask` 字段用于存放掩码,它表示在已经打开的文件上要设置的访问权限。

3. `root`、`pwd`和`altpwd`

`root` 字段和 `pwd` 字段是两个指针,分别指向与进程的根目录、当前工作目录相关的目录项对象。`Altpwd` 也是一个指针,指向可选根目录的目录项结构。这个字段用于仿真环境。

4. `rootmnt`、`pwdmnt`和`altpwrmnt`

`Rootmnt`、`pwdmnt` 和 `altpwrmnt` 三个指针,分别指向进程根目录、当前工作目录和可选根目录所对应的安装文件系统对象。

6.3.2 `files_struct` 结构

`files_struct` 包含与打开的文件及其描述符相关的信息。在引言中提到,文件描述符是和打开文件相关的、唯一的整型数据。用内核术语来说,文件描述符就是当前进程的 `task_struct` 的文件对象的 `fd` 数组的下标,也就是 `current->files->fd`。图 6-7 说明了 `task_struct` 的 `fd` 数组,以及它是如何指向文件的文件结构的。

根据共享特性,例如只读或读/写, Linux 可以关联文件描述符的集合。`fd_struct` 结构体表示文件描述符集。`files_struct` 使用这些集合来对它的文件描述符进行分组。

```
-----
include/linux/file.h
22 struct files_struct {
23     atomic_t count;
24     spinlock_t file_lock;
25     int max_fds;
26     int max_fdset;
27     int next_fd;
28     struct file ** fd;
29     fd_set *close_on_exec;
30     fd_set *open_fds;
31     fd_set close_on_exec_init;
-----
```

```

32     fd_set open_fds_init;
33     struct file * fd_array[NR_OPEN_DEFAULT];
34 };

```

第 23 行：与 `fs_struct` 类似，由于 `files_struct` 能够被多个进程描述符引用，所以必须有 `count` 字段。这个字段在内核例程 `fget()` 中增加，而在内核例程 `fput()` 中减小。在文件关闭期间调用这两个函数。

第 25 行：`max_fds` 字段记录进程最多能够打开的文件数。与 `fd_array` 的默认大小 `NR_OPEN_DEFAULT` 有关，`max_fds` 的默认值是 32。当进程想要打开的文件多于 32 个时，就要增加 `max_fds` 的值。

第 26 行：`max_fdset` 字段记录文件描述符的最大数。类似于 `max_fds`，如果进程打开的文件总数超过了这个值，可以增加这个字段的值。

第 27 行：`next_fd` 字段存放下一个将要分配的文件描述符的值。我们看看如何通过文件的打开和关闭来操作它，但是应该清楚一件事：文件描述符是以递增方式分配的，如果先前分配的文件描述符所对应的文件被关闭，那么，`next_fd` 字段就会被设置成被关闭的那个描述符的值。因此，文件描述符是按照最低可用值的方式进行分配的。

第 28 行：`fd` 数组指向打开的文件对象的数组。它默认指向 `fd_array`，`fd_array` 可以存放 32 个描述符。当出现一个大于 32 的文件描述符的请求时，它指向一个新生成的数组。

第 30~32 行：`close_on_exec`、`open_fds`、`close_on_exec_init` 和 `open_fds_init` 都是 `fd_set` 类型的字段。前面已经提到，`fd_set` 结构体用于存放文件描述符集。在对每个字段单独解释之前，我们先来看看 `fd_set` 结构体。

`Fd_set` 数据类型可被追溯到一个存放无符号长整型数组的结构体——数组的每个元素都存放一个文件描述符。

```

-----
include/linux/types.h
22  typedef __kernel_fd_set  fd_set;
-----

```

`fd_set` 数据类型就是 `__kernel_fd_set` 类型。这种数据类型的结构体用于存放无符号长整型数组。

```

-----
include/linux/posix_types.h
36  typedef struct {
37     unsigned long fds_bits [__FDSET_LONGS];
38 } __kernel_fd_set;
-----

```

`__FDSET_LONGS` 在 32 位系统中的值是 32，而在 64 位系统中的值是 16，这确保 `fd_sets` 总是拥有 1024 大小的位图。此处定义了 `__FDSET_LONGS`：

```

-----
include/linux/posix_types.h
6  #undef __NFDBITS

```

```

7  #define __NFDBITS (8 * sizeof(unsigned long))
8
9  #undef __FD_SETSIZE
10 #define __FD_SETSIZE 1024
11
12 #undef __FDSET_LONGS
13 #define __FDSET_LONGS (__FD_SETSIZE/__NFDBITS)
-----

```

可以用 4 个宏来操作这些文件描述符的集合（参见表 6-8）。

表6-8 文件描述符集的宏

宏	说 明
FD_SET	在集合中设置文件描述符
FD_CLR	从集合中清除文件描述符
FD_ZERO	清除文件描述符集
FD_ISSET	如果文件描述符被设置，则返回

现在，我们来看看各个字段。

1. close_on_exec

close_on_exec 字段是指向一个文件描述符集的指针，这些文件描述符标记为“执行 exec()时关闭”。close_on_exec 初始化为（并且通常）指向 close_on_exec_init 字段。如果被标记为“执行 exec()时关闭”的文件描述符数超出 close_on_exec_init 位字段大小时，则修改 close_on_exec 字段。

2. open_fds

open_fds 字段是指向文件描述符集的指针，这些文件描述符被标记为“打开”。和 close_on_exec 类似，它最开始指向 open_fds_init，并且如果被标记为“打开”的文件描述符数超出 open_fds_init 位字段的大小时，则修改 open_fds 字段。

3. close_on_exec_init

close_on_exec_init 字段存放位字段，用于记录在执行 exec()时要关闭的文件的文件描述符。

4. open_fds_init

open_fds_init 字段存放位字段，用于记录被打开的文件对应的文件描述符。

5. fd_array

fd_array 指针数组指向前 32 个打开的文件描述符。

下面通过宏 INIT_FILES 来初始化 fs_struct 结构体：

```

-----
include/linux/init_task.h
6  #define INIT_FILES \
7  {
8    .count      = ATOMIC_INIT(1),
9    .file_lock  = SPIN_LOCK_UNLOCKED,

```

```

10 .max_fds    = NR_OPEN_DEFAULT,
11 .max_fdset  = __FD_SETSIZE,
12 .next_fd    = 0,
13 .fd         = &init_files.fd_array[0],
14 .close_on_exec = &init_files.close_on_exec_init,
15 .open_fds    = &init_files.open_fds_init,
16 .close_on_exec_init = { { 0, } },
17 .open_fds_init = { { 0, } },
18 .fd_array    = { NULL, }
19 }

```

图 6-11 给出了初始化后的 fs_struct。

```

include/linux/file.h
6  #define NR_OPEN_DEFAULT BITS_PER_LONG

```

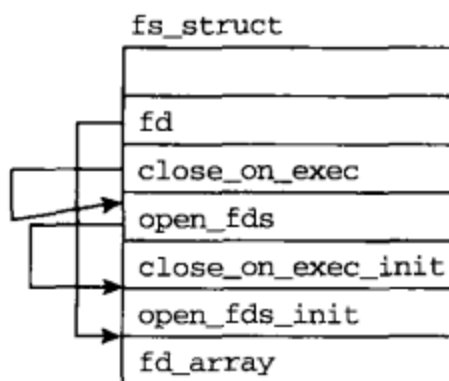


图 6-11 初始化后的 fs_struct

NR_OPEN_DEFAULT 的全局定义被设置为 BITS_PER_LONG。BITS_PER_LONG 在 32 位的系统中是 32，在 64 位的系统中是 64。

6.4 页缓存

在引言中，我们提到页缓存（Page Cache）是内存中页的集合。当频繁访问数据时，能够快速访问数据是至关重要的。当在两个设备之间复制和同步数据时，其中的一个设备通常在存储量方面小于另一个设备，但是却允许更快地访问，我们把这个存储量小但速度快的设备叫做缓存（Cache）。页缓存就是操作系统在内存中存放部分硬盘数据以加快访问的一种方式。现在，我们来看看其工作和实现机理。

对硬盘上的文件执行写操作时，那个文件被分割成称为页的块，这些块被换入内存（RAM）。操作系统修改内存中的页，随后，更新的页被写入磁盘。

如果页从硬盘复制到 RAM（这被称为换入内存），它可能是干净的，也可能变脏。脏页在内存中被修改，但是这个修改还没有写到磁盘上。干净的页在内存中的状态和在磁盘上的状态完全一致。

在 Linux 中，内存被分成区^①。每个区拥有活跃页的链表和非活跃页的链表。如果某个页面已经有一段时间处于不活跃状态，它就会被换出（写回磁盘），从而释放内存。区链表中的每个页均有一个指向 `address_space` 结构体的指针。每个 `address_space` 有一个指向 `address_space_operations` 结构体的指针。通过调用 `address_space_operation` 结构体的 `set_dirty_page()` 函数把页标记为脏。图 6-12 说明了这种依赖关系。

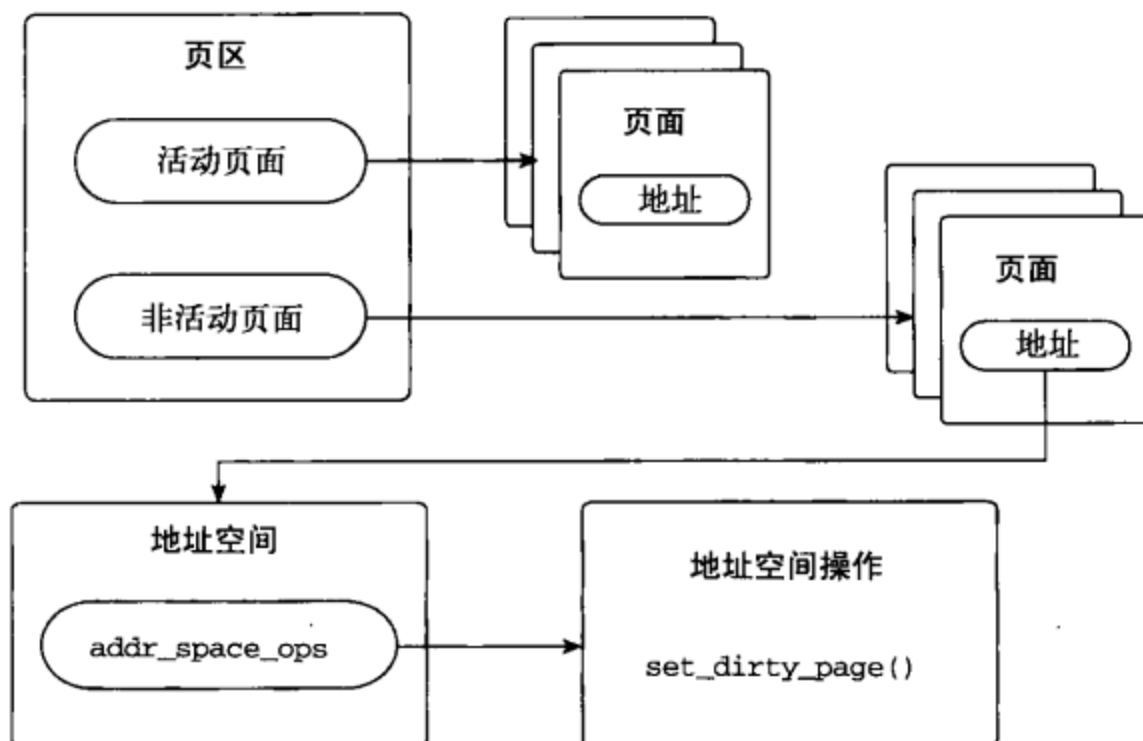


图 6-12 页缓存和区

6.4.1 address_space 结构

页缓存的核心是 `address_space` 对象。让我们来详细分析这个对象：

```

-----
include/linux/fs.h
326 struct address_space {
327     struct inode *host; /* owner: inode, block_device */
328     struct radix_tree_root page_tree; /* radix tree of all pages */
329     spinlock_t tree_lock; /* and spinlock protecting it */
330     unsigned long nrpages; /* number of total pages */
331     pgoff_t writeback_index; /* writeback starts here */
332     struct address_space_operations *a_ops; /* methods */
333     struct prio_tree_root i_mmap; /* tree of private mappings */
334     unsigned int i_mmap_writable; /* count VM_SHARED mappings */
335     struct list_head i_mmap_nonlinear; /* list VM_NONLINEAR mappings */
336     spinlock_t i_mmap_lock; /* protect tree, count, list */
337     atomic_t truncate_count; /* Cover race condition with truncate */
338     unsigned long flags; /* error bits/gfp mask */
339     struct backing_dev_info *backing_dev_info; /* device readahead, etc */
340     spinlock_t private_lock; /* for use by the address_space */
  
```

① 关于内存区的更多信息参见第 4 章。


```

341 struct list_head private_list; /* ditto */
342 struct address_space *assoc_mapping; /* ditto */
343 };

```

结构体中内联的注释已经描述得相当清楚。一些附加的说明也许有助于理解如何操作页缓存。

通常, `address_space` 关联索引节点以及指向这个索引节点的 `host` 字段。不过, 一般来说, 页缓存和地址空间结构并不需要这个字段。如果 `address_space` 对应的内核对象不是索引节点, 那么它可能为空。

`address_space` 结构体有一个直观上让人非常熟悉的字段: `address_space_operations`。与文件结构的 `file_operations` 非常类似, `address_space_operations` 包含 `address_space` 的有效操作的相关信息。

```

-----
include/linux/fs.h
297 struct address_space_operations {
298     int (*writepage)(struct page *page, struct writeback_control *wbc);
299     int (*readpage)(struct file *, struct page *);
300     int (*sync_page)(struct page *);
301
302     /* Write back some dirty pages from this mapping. */
303     int (*writepages)(struct address_space *, struct writeback_control *);
304
305     /* Set a page dirty */
306     int (*set_page_dirty)(struct page *page);
307
308     int (*readpages)(struct file *filp, struct address_space *mapping,
309                     struct list_head *pages, unsigned nr_pages);
310
311     /*
312     * ext3 requires that a successful prepare_write() call be followed
313     * by a commit_write() call - they must be balanced
314     */
315     int (*prepare_write)(struct file *, struct page *, unsigned, unsigned);
316     int (*commit_write)(struct file *, struct page *, unsigned, unsigned);
317     /* Unfortunately this kludge is needed for FIBMAP. Don't use it */
318     sector_t (*bmap)(struct address_space *, sector_t);
319     int (*invalidatepage)(struct page *, unsigned long);
320     int (*releasepage)(struct page *, int);
321     ssize_t (*direct_IO)(int, struct kiocb *, const struct iovec *iov,
322                          loff_t offset, unsigned long nr_segs);
323 };
-----

```

这些函数的意义不言自明。`readpage()`和 `writepage()`分别读写与地址空间相关的页。利用 `readpages()`和 `writepages()`可以读写多个页。日志文件系统(例如 `ext3`)可以提供 `prepare_write()`和 `commit_write()`函数。

当内核在页缓存中查找一个页时, 它必须快速查找。同样地, 每个地址空间有一个 `radix_tree`, 它完成一个快速的搜索, 以确定查找的页是否在页缓存中。

图 6-13 说明了文件、索引节点、地址空间以及页之间是如何互相关联的；这副图对于即将进行的页缓存代码的分析很有用处。

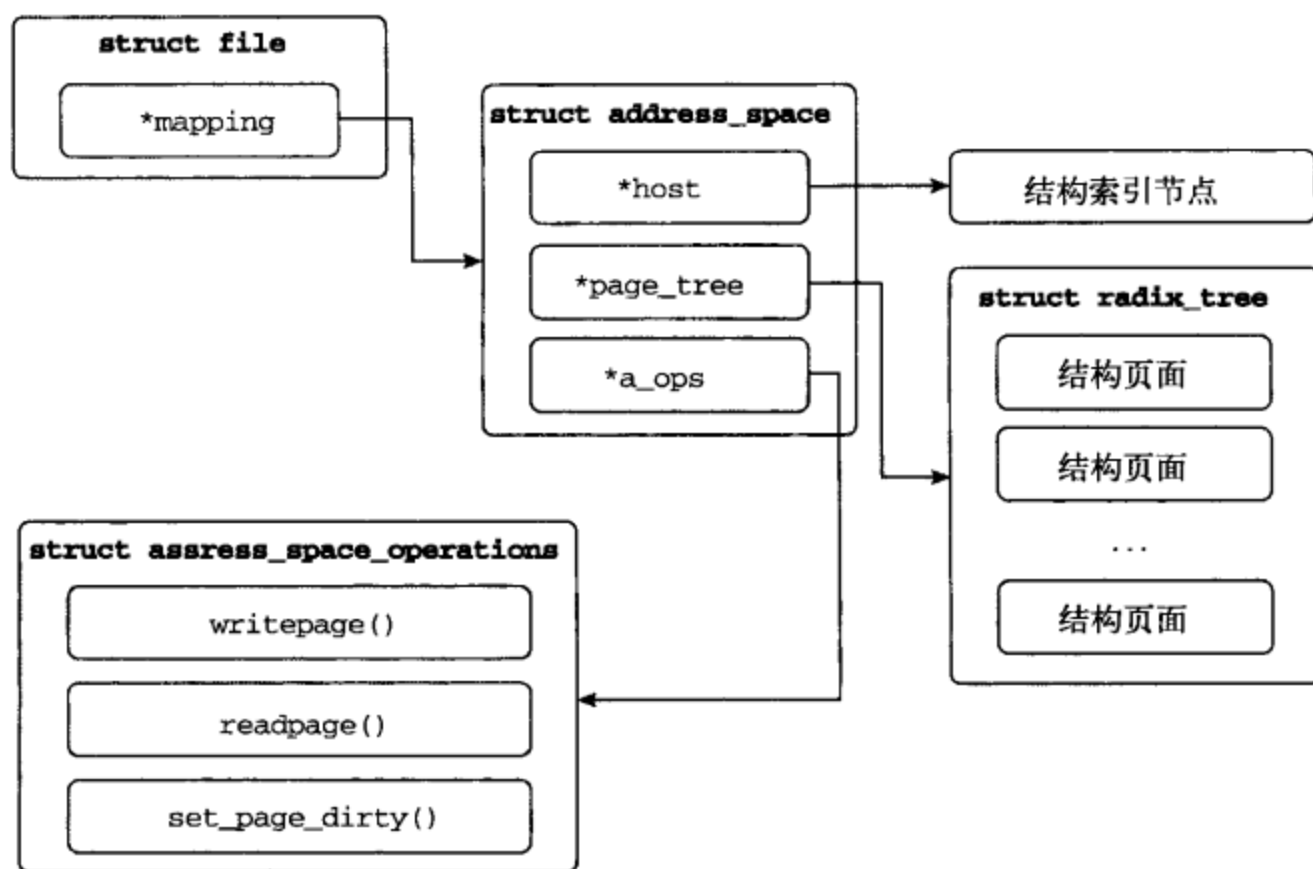


图 6-13 文件、索引节点、地址空间和页

6.4.2 buffer_head 结构

Linux 内核把块设备上的每个扇区表示为 **buffer_head** 结构体。**buffer_head** 包含把物理扇区映射成物理内存中一个缓冲区所有必要的信息。图 6-14 中说明了 **buffer_head** 结构体。

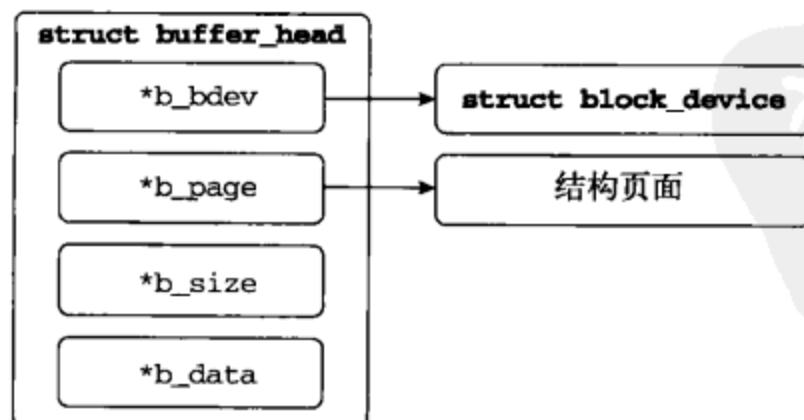


图 6-14 buffer_head 结构体

```

-----
include/linux/buffer_head.h
47 struct buffer_head {
48     /* First cache line: */
49     unsigned long b_state; /* buffer state bitmap (see above) */
50     atomic_t b_count; /* users using this block */

```

```

51 struct buffer_head *b_this_page; /* circular list of page's buffers */
52 struct page *b_page; /* the page this bh is mapped to */
53
54 sector_t b_blocknr; /* block number */
55 u32 b_size; /* block size */
56 char *b_data; /* pointer to data block */
57
58 struct block_device *b_bdev;
59 bh_end_io_t *b_end_io; /* I/O completion */
60 void *b_private; /* reserved for b_end_io */
61 struct list_head b_assoc_buffers; /* associated with another mapping */
62 };

```

buffer_head 结构体所涉及的物理扇区是设备 b_dev 上的逻辑块 b_blocknr。

buffer_head 结构体所涉及的物理内存是起始地址为 b_data 而块大小为 b_size 字节的内存块。这个内存块在物理页 b_page 中。

buffer_head 结构体中的其他定义用于管理物理扇区如何映射到物理内存这样的事务。(因为这牵扯到了 bio 结构体，而不仅仅是 buffer_head，所以，有关结构体 buffer_head 更详细的信息请参见 mpage.c)。

第4章中提到，Linux 内核中每个物理内存页由页结构体表示。页由许多 I/O 块组成。因为每个 I/O 块可能没有页那么大（尽管块可能更小），所以页是由一个或多个 I/O 块组成的。

在较早的 Linux 版本中，块的 I/O 只能利用缓冲区来完成，但在 Linux 2.6 中，一个新方法被开发出来，那就是利用 bio 结构。这个新方法允许 Linux 内核用一种更容易管理的方式把块 I/O 组织在一起。

假设我们要写一个文本文件的开始部分和结束部分。这个更改可能需要两个 buffer_head 结构体用于数据传送：一个指向开始部分，一个指向结束部分。bio 结构允许文件操作仅用一个结构体把不连续的块集合到一起。这种查看缓冲和页的新方法发生在查看缓冲区 (buffer) 的连续内存段时。bio_vec 结构体表示缓冲区中的连续内存段。bio_vec 结构体如图 6-15 所示。

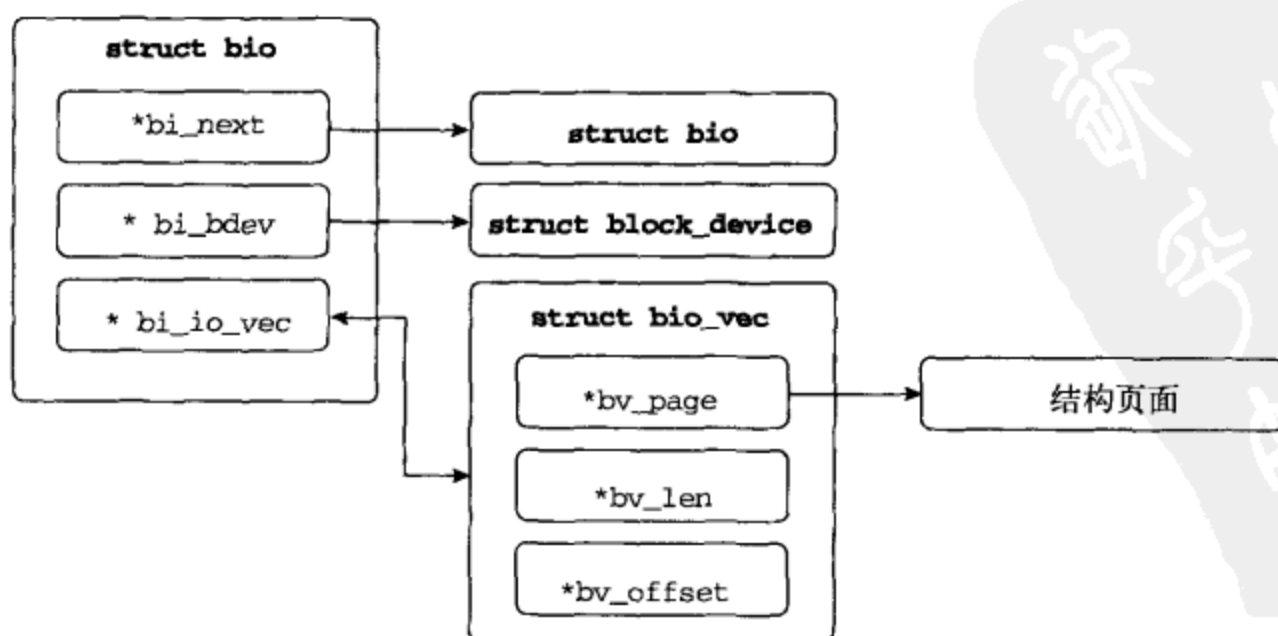


图 6-15 bio 结构

```

-----
include/linux/bio.h
47 struct bio_vec {
48     struct page *bv_page;
49     unsigned int bv_len;
50     unsigned int bv_offset;
51 };
-----

```

bio_vec 结构体存放的信息包括：一个指向页的指针、内存段的长度、段在页中的偏移量。

bio 结构由一个 bio_vec 结构体数组组成（连同其他的管理字段）。因此，bio 结构表示一个或多个缓冲区的多个连续的内存段，这些缓冲区又属于一个或多个页^①。

6.5 VFS 的系统调用和文件系统层

至此，我们介绍了与 VFS 相关的所有结构以及页缓存。接下来，我们将集中精力研究用于文件操作的两个系统调用，并追踪它们的执行直到内核层面。我们来看看 open()、close()、read() 以及 write() 系统调用如何利用先前所描述的那些结构。

之前已经提到，在 VFS 中，文件被视为纯抽象的概念。你能够打开、读、写、关闭一个文件，但是实际发生的细节对 VFS 层并不重要。第 5 章介绍了这些细节。

与 VFS 对应的是文件系统特有的层，这个层把 VFS 的文件 I/O 转换成页和块。因为在计算机系统中你可以拥有许多具体的文件系统类型，例如 ext2 格式的硬盘和 iso9660 的 cdrom，所以，文件系统层可被分成两个主要部分：通用文件系统的操作和具体文件系统的操作（参见图 6-3）。

依照自上而下的方法，本节追踪从 VFS 调用 read() 或 write() 而发出的读写请求着手，穿过文件系统层，一直到具体的块 I/O 请求被提交给块设备驱动程序为止。在这个过程中，我们从通用文件系统层转移到特定文件系统层。在此，以 ext2 文件系统驱动程序作为特定文件系统层的一个实例，但是要切记：访问哪个文件系统的驱动程序取决于正在操作的文件。当我们进行研究时，也会遇到页缓存，它是位于 Linux 通用文件系统层的一种结构。在较早的 Linux 版本中，有缓冲区缓存和页缓存，但在 2.6 内核中，页缓存代替了缓冲区缓存的所有功能。

6.5.1 open()

当进程想要读写文件内容时，它就发出 open() 系统调用：

```

-----
synopsis
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
-----

```

open 系统调用的参数包括：文件的路径名、标识所打开文件访问模式的标志和权限的位掩码

^① 关于 bio 结构的更多信息参见 include/linux/bio.h。

(文件是否被创建)。open()返回所打开的文件的文件描述符(如果打开成功)或错误码(如果打开失败)。

flags 参数是通过按位或一个或者多个常量来形成的,这些常量在 include/linux/fcntl.h 中定义。表 6-9 列出了 open()使用的标志,以及相应的常量值。O_RDONLY、O_WRONLY 或 O_RDWR 标志只能指定其中一种。其他的标志是可选的。

表6-9 open()的标志

标志名	值	说明
O_RDONLY	0	以只读方式打开文件
O_WRONLY	1	以只写方式打开文件
O_RDWR	2	以读写方式打开文件
O_CREAT	100	表示如果文件不存在,则创建该文件。设置了此标志的open()函数等同于creat()函数
O_EXCL	200	与O_CREAT一起使用,这个标志表示如果文件不存在,则open()失败
O_NOCTTY	400	在路径名引用终端设备的情况中,进程不应该把它当做控制终端
O_TRUNC	0x1000	如果文件存在,把它截为空
O_APPEND	0x2000	从文件末开始写
O_NONBLOCK	0x4000	以非阻塞模式打开文件
O_NDELAY	0x4000	和O_NONBLOCK的值相同
O_SYNC	0x10000	文件的写操作必须等待物理I/O的完成。应用于块设备上的文件
O_DIRECT	0x20000	对文件操作,让I/O上的高速缓存达到最小化
O_LARGEFILE	0x100000	大文件系统允许的文件长度大于31位表示的文件长度。这个标志确保可以打开大文件
O_DIRECTORY	0x200000	如果路径名表示的不是目录,则打开失败
O_NOFOLLOW	0x400000	如果路径名是符号链,则打开失败

让我们来看看系统调用:

```
-----
fs/open.c
927  asmlinkage long sys_open (const char __user * filename, int flags, int mode)
928  {
929      char * tmp;
930      int fd, error;
931
932      #if BITS_PER_LONG != 32
933          flags |= O_LARGEFILE;
934      #endif
935      tmp = getname(filename);
936      fd = PTR_ERR(tmp);
937      if (!IS_ERR(tmp)) {
938          fd = get_unused_fd();
939          if (fd >= 0) {
940              struct file *f = filp_open(tmp, flags, mode);
941              error = PTR_ERR(f);
942              if (IS_ERR(f))
```

```

943     goto out_error;
944     fd_install(fd, f);
945 }
946 out:
947     putname(tmp);
948 }
949     return fd;
950
951 out_error:
952     put_unused_fd(fd);
953     fd = error;
954     goto out;
955 }

```

第 932~934 行：检查我们的系统是否是非 32 位的。如果不是 32 位的，就启用大文件系统的支持标志 `O_LARGEFILE`。这使得 `sys_open` 函数能够打开的文件，其长度大于 31 位表示的文件长度。

第 935 行：`getname()` 例程通过调用 `strncpy_from_user()` 把文件名从用户空间复制到内核空间。

第 938 行：`get_unused_fd()` 例程返回第一个可用的文件描述符（或者说 `fd` 数组的索引：`current->files->fd`），并把它标记为忙。局部变量 `fd` 被设置成这个值。

第 940 行：`filp_open()` 函数完成 `open` 系统调用的大部分工作，并且返回关联进程和文件的文件结构体。接下来详细分析 `filp_open()` 例程：

```

-----
fs/open.c
740 struct file *filp_open(const char * filename, int flags, int mode)
741 {
742     int namei_flags, error;
743     struct nameidata nd;
744
745     namei_flags = flags;
746     if ((namei_flags+1) & O_ACCMODE)
747         namei_flags++;
748     if (namei_flags & O_TRUNC)
749         namei_flags |= 2;
750
751     error = open_namei(filename, namei_flags, mode, &nd);
752     if (!error)
753         return dentry_open(nd.dentry, nd.mnt, flags);
754
755     return ERR_PTR(error);
}
-----

```

第 745~749 行：这是路径名查找函数，例如 `open_namei()`，其访问模式标志的编码不同于 `open` 系统调用，具有其特有的格式。这几行将访问模式标志复制给 `namei_flags` 变量，并将该标志格式化为可以被 `open_namei()` 解释的访问模式标志。

路径名查找的主要差异可能在于：访问模式也许并没有要求读写权限。但是当试图打开文件

时,这种“无权限”的访问模式没有任何意义,因此 open 系统调用的标志不包括“无权限”。“无权限”用值 00 来表示。“读权限”是通过把低位的值设置为 1 来表示的,而“写权限”是通过把高位的值设置为 1 来表示的。在 include/asm/fcntl.h 中,open 系统调用的标志 O_RDONLY、O_WRONLY 和 O_RDWR 对应的值分别是 00、01 和 02。

namei_flags 变量能够通过把自己和 O_ACCMODE 常量进行逻辑按位与来获得访问模式。O_ACCMODE 的值是 3,如果要和 O_ACCMODE 进行按位与的变量值是 1、2 或 3,计算的结果就为真。因此,如果把 open 系统调用的标志设置成 O_RDONLY、O_WRONLY 和 O_RDWR,给这个值加 1 就可以把它转换成路径名查找格式,这样,当它和 O_ACCMODE “与”的时候,结果就为真。第二次检查仅仅是为了确保当把 open 系统调用标志设置为允许文件截断时,就在访问模式中把高位设置为 1,以表明允许写操作。

第 751 行: open_namei() 例程完成路径名的查找,生成相关的 nameidata 结构体并取得相应的索引节点。

第 753 行: dentry_open() 是围绕 dentry_open_it() 的封装例程, dentry_open_it() 创建并初始化文件结构体。创建文件结构体是通过调用内核例程 get_empty_filp() 实现的。如果 files_stat.nr_files 大于或等于 files_stat.max_files, 该例程就返回 ENFILE, 这种情形表示打开的文件总数已达到系统的最大限制。

接下来分析 dentry_open_it() 例程:

```
-----
fs/open.c
844 struct file *dentry_open_it(struct dentry *dentry, struct 845 vfsmount *mnt, int
    flags, struct lookup_intent *it)
846 {
847     struct file * f;
848     struct inode *inode;
849     int error;
850
851     error = -ENFILE;
852     f = get_empty_filp();
853     ...
855     f->f_flags = flags;
856     f->f_mode = (flags+1) & O_ACCMODE;
857     f->f_it = it;
858     inode = dentry->d_inode;
859     if (f->f_mode & FMODE_WRITE) {
860         error = get_write_access(inode);
861         if (error)
862             goto cleanup_file;
863     }
864     ...
866     f->f_dentry = dentry;
867     f->f_vfsmnt = mnt;
868     f->f_pos = 0;
869     f->f_op = fops_get(inode->i_fop);
870     file_move(f, &inode->i_sb->s_files);
871
872     if (f->f_op && f->f_op->open) {
```

```

873     error = f->f_op->open(inode,f);
874     if (error)
875         goto cleanup_all;
876     intent_release(it);
877 }
...
891 return f;
...
907 }
-----

```

第 852 行：调用 `get_empty_filp()` 来分配文件结构体。

第 855~856 行：用传入 `open` 系统调用的标志设置文件结构体的 `f_flags` 字段。把传给 `open` 系统调用的访问模式先转换成路径名查找函数所需的格式，然后用它来设置文件结构体的 `f_mode` 字段。

第 866~869 行：把文件结构体的 `f_dentry` 字段设置成指向与文件路径名有关的目录项结构体。把 `f_vfsmnt` 字段设置成指向具体文件系统的 `vmfsmount` 结构体。`f_pos` 被设置为 0，表示 `file_offset` 的起始位置位于文件的开头。`f_op` 字段被设置成指向由文件索引节点所指向的操作表。

第 870 行：调用 `file_move()` 例程把已打开的文件的文件结构体插入其所在文件系统的超级块链表中。

第 872~877 行：此处，调用 `open` 函数的下层函数。如果文件中还有多个特定于文件的功能要执行，以打开文件，则调用这个打开函数。如果文件的操作表中含有打开例程，也调用这样的打开例程。

至此，已经涵盖了 `dentry_open_it()` 例程的所有内容。

到 `filp_open()` 结束时，将刚分配的文件结构体插入超级块 `s_files` 字段的头部，同时让 `f_dentry` 指向 `dentry` 对象、`f_vfsmount` 指向 `vfsmount` 对象、`f_op` 指向索引节点的文件操作表 `i_fop`，并把 `f_flags` 设置成访问标志，把 `f_mode` 设置成传给 `open()` 调用的权限模式。

第 944 行：`fd_install()` 例程设置 `fd` 数组指针，使之指向由 `filp_open()` 返回的文件对象的地址。换句话说，就是设置 `current->files->fd[fd]`。

第 947 行：`putname()` 例程释放文件名所占用的内核空间。

第 949 行：返回文件描述符 `fd`。

第 952 行：`put_unused_fd()` 例程清除已经分配的描述符。当文件对象创建失败时调用该例程。

总之，`open()` 系统调用的多级调用过程看起来是下面这样的。

`sys_open`:

- `getname()`：把文件名传送到内核空间；
- `get_unused_fd()`：获得下一个可用的文件描述符；
- `filp_open()`：创建 `nameidata` 结构体；

- `open_namei()`: 初始化 `nameidata` 结构体;
- `dentry_open()`: 创建并初始化文件对象;
- `fd_install()`: 将 `current->files->fd[fd]` 设置成文件对象;
- `putname()`: 回收为文件名分配的内核空间。

图 6-16 说明了被初始化和设置的结构, 并标识了完成这些初始化和设置的例程。

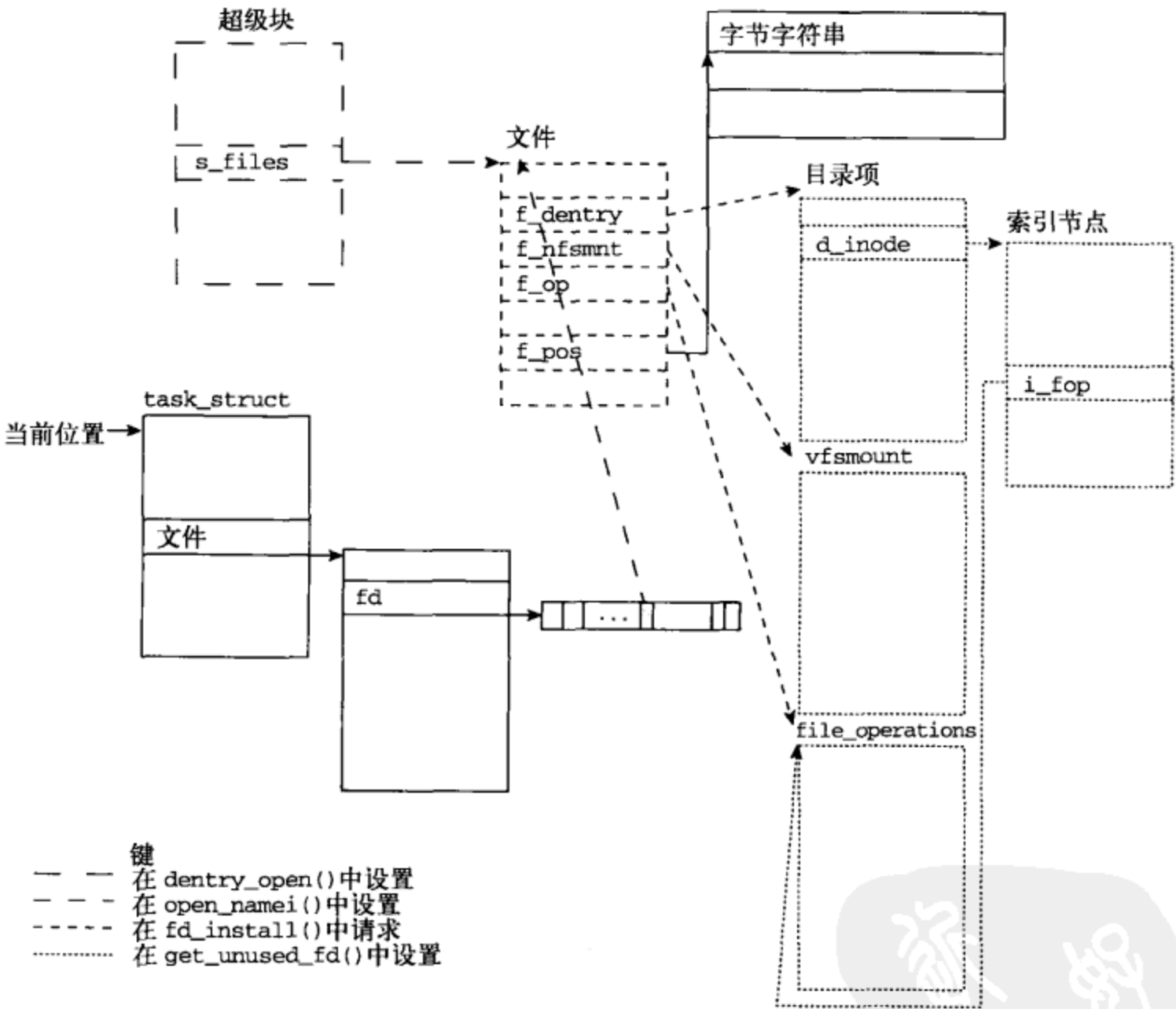


图 6-16 文件系统的结构

表 6-10 给出了 `sys_open()` 返回的一些错误以及找到这些错误的内核例程。

表6-10 `sys_open()` 返回的错误

错 误 码	说 明	返回错误的函数
ENAMETOOLONG	路径名太长	<code>getname()</code>
ENOENT	文件不存在 (标志 <code>O_CREAT</code> 也没有设置)	<code>getname()</code>
EMFILE	进程打开的文件数达到最大值	<code>get_unused_fd()</code>
ENFILE	系统中打开的文件数达到最大值	<code>get_unused_filp()</code>

6.5.2 close()

进程使用完文件后，就发出 close() 系统调用：

```
-----  
synopsis  
#include <unistd.h>  
  
int close(int fd);  
-----
```

close 系统调用把将要关闭的文件的文件描述符作为参数。在标准 C 程序中，在程序终止时隐含着对 close() 的调用。让我们深入分析一下 sys_close() 的代码：

```
-----  
fs/open.c  
1020 asmlinkage long sys_close(unsigned int fd)  
1021 {  
1022     struct file * filp;  
1023     struct files_struct *files = current->files;  
1024  
1025     spin_lock(&files->file_lock);  
1026     if (fd >= files->max_fds)  
1027         goto out_unlock;  
1028     filp = files->fd[fd];  
1029     if (!filp)  
1030         goto out_unlock;  
1031     files->fd[fd] = NULL;  
1032     FD_CLR(fd, files->close_on_exec);  
1033     __put_unused_fd(files, fd);  
1034     spin_unlock(&files->file_lock);  
1035     return filp_close(filp, files);  
1036  
1037 out_unlock:  
1038     spin_unlock(&files->file_lock);  
1039     return -EBADF;  
1040 }  
-----
```

第 1023 行：当前 task_struct 的 files 字段指向与我们文件对应的 files_struct。

第 1025~1030 行：这几行首先给文件加锁，以免遇到同步问题。然后，检查文件描述符是否有效。如果文件描述符编号大于这个文件所允许的最大值，就删除锁并返回错误码-EBADF。否则将获得文件结构体的地址。如果文件描述符索引不能产生一个文件结构体，我们也删除锁并返回错误，因为没有文件要关闭。

第 1031~1032 行：此处，将 current->files->fd[fd] 设置为 NULL，即删除指向文件对象的指针。同时也在 files->close_on_exec 所指向的文件描述符集中清除该文件描述符的相应位。因为文件描述符被关闭，所以进程不必担心在调用 exec() 时还会涉及它。

第 1033 行：因为这个文件不再是打开的，内核例程 __put_unused_fd() 在文件描述符集 files->open_fds 中清除这个文件描述符的相应位。__put_unused_fd() 采取了一些措施，以确保遵循文件描述符的“最小可用索引”的分配原则：

```

-----
fs/open.c
897 static inline void __put_unused_fd(struct files_struct *files, unsigned int fd)
898 {
899     __FD_CLR(fd, files->open_fds);
900     if (fd < files->next_fd)
901         files->next_fd = fd;
902 }
-----

```

第 890~891 行：next_fd 字段保存下一个将要分配的文件描述符的值。如果当前文件描述符的值小于 files->next_fd 中的值，则把这个域设置为当前文件描述符的值。这样做可以保证文件描述符总是遵循“最小可用值”的原则进行分配。

第 1034~1035 行：现在释放文件锁，并且把控制权传递给 filp_close() 函数，该函数负责为 close 系统调用返回合适的值。filp_close() 函数完成 close 系统调用的大部分工作。接下来详细分析一下 filp_close() 例程：

```

-----
fs/open.c
987 int filp_close(struct file *filp, fl_owner_t id)
988 {
989     int retval;
990     /* Report and clear outstanding errors */
991     retval = filp->f_error;
992     if (retval)
993         filp->f_error = 0;
994
995     if (!file_count(filp)) {
996         printk(KERN_ERR "VFS: Close: file count is 0\n");
997         return retval;
998     }
999
1000     if (filp->f_op && filp->f_op->flush) {
1001         int err = filp->f_op->flush(filp);
1002         if (!retval)
1003             retval = err;
1004     }
1005
1006     dnotify_flush(filp, id);
1007     locks_remove_posix(filp, id);
1008     fput(filp);
1009     return retval;
1010 }
-----

```

第 991~993 行：这几行清除任何明显的错误。

第 995~997 行：这是关闭文件时进行完整性检查的必要条件。file_count 为 0 的文件应该早已关闭。因此，在这种情况下，filp_clos 将返回一个错误。

第 1000~1001 行：调用文件操作函数 flush()（如果定义了的话）。它的功能取决于具体的文件系统。

第 1008 行：调用 `fput()` 来释放文件结构体。这个例程完成的操作包括：调用文件操作函数 `release()`，删除指向目录项对象和 `vfsmount` 对象的指针，最后，释放文件对象。

`close()` 系统调用的多级调用过程如下。

sys_close()：

□ `__put_unused_fd()`：把文件描述符归还给可用池。

□ `filp_close()`：准备即将清除的文件对象。

□ `fput()`：清除文件对象。

表 6-11 给出了 `sys_close()` 返回的一些错误以及找到这些错误的内核例程。

表6-11 `sys_close()`返回的错误

错 误	函 数	说 明
EBADF	<code>sys_close()</code>	无效的文件描述符

6.5.3 read()

当用户级程序调用函数 `read()` 时，Linux 把它转换成系统调用 `sys_read()`：

```
-----
fs/read_write.c
272 asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf, size_t count)
273 {
274     struct file *file;
275     ssize_t ret = -EBADF;
276     int fput_needed;
277
278     file = fget_light(fd, &fput_needed);
279     if (file) {
280         ret = vfs_read(file, buf, count, &file->f_pos);
281         fput_light(file, fput_needed);
282     }
283
284     return ret;
285 }
-----
```

第 272 行：`sys_read()` 用于获取文件描述符、用户空间的缓冲区指针以及从文件读入缓冲区的字节数。

第 273~282 行：文件的查找是通过 `fget_light()` 把文件描述符转换成文件指针来完成的。然后调用 `vfs_read()` 以完成所有主要的工作。每个 `fget_light` 都必须和 `fput_light()` 成对出现，因此完成 `vfs_read()` 后就调用 `fput_light()`。

系统调用 `sys_read()` 把控制权传给 `vfs_read()`，因此，继续追踪 `vfs_read()` 吧：

```
-----
fs/read_write.c
200 ssize_t vfs_read(struct file *file, char __user *buf, size_t count,
loff_t *pos)
201 {
```



```

202 struct inode *inode = file->f_dentry->d_inode;
203 ssize_t ret;
204
205 if (!(file->f_mode & FMODE_READ))
206     return -EBADF;
207 if (!file->f_op || (!file->f_op->read && \                                !file->f_op->aio_read))
208     return -EINVAL;
209
210 ret = locks_verify_area(FLOCK_VERIFY_READ, inode,
211 file, *pos, count);
212 if (!ret) {
213     ret = security_file_permission (file, MAY_READ);
214     if (!ret) {
215         if (file->f_op->read)
216             ret = file->f_op->read(file,
217 buf, count, pos);
218         else
219             ret = do_sync_read(file, buf,
220 count, pos);
221         if (ret > 0)
222             dnotify_parent(file->f_dentry,
223 DN_ACCESS);
224     }
225 }
226 return ret;
227 }

```

第 200 行：前 3 个参数都是经由 `sys_read()` 的参数传入（或者转换而成）的。第四个参数是文件内部的偏移，读操作从该偏移处开始。如果显式地调用 `vfs_read()`，文件偏移可能不为 0，因为 `vfs_read()` 可能是从内核内部被调用的。

第 202 行：保存指向文件索引节点的指针。

第 205~208 行：对文件操作结构进行基本检查，确保已经定义了读操作或异步读操作。如果没有定义读操作，或者找不到操作表，该函数就在此处返回 `EINVAL` 错误。这个错误表示文件描述符所关联的结构体不能进行读操作。

第 210~214 行：检验要读的区域是不是没有上锁，并检验文件是不是已经被授权读。只要这两个条件不同时满足，就通知该文件的父目录（见第 218~219 行）。

第 215~217 行：这是 `vfs_read()` 的主要内容。如果已经定义了读文件操作，就调用它；否则，调用 `do_sync_read()`。

在追踪过程中，我们追踪的是标准的文件操作 `read()` 而不是 `do_sync_read()`。随后，可以清楚地看到，在底层，它们最终殊途同归。

1. 从通用文件系统层到特定文件系统层

从通用文件系统层过渡到特定文件系统层，这是在众多抽象概念中我们遇到的第一个。图 6-17 说明了文件结构体是如何指向特定文件系统的表或操作的。回想一下，当调用 `read_inode()` 时，索引节点的信息被填充，其中就包括让 `fop` 字段指向由特定文件系统（例如 `ext2`）所定义的合适的操作表。

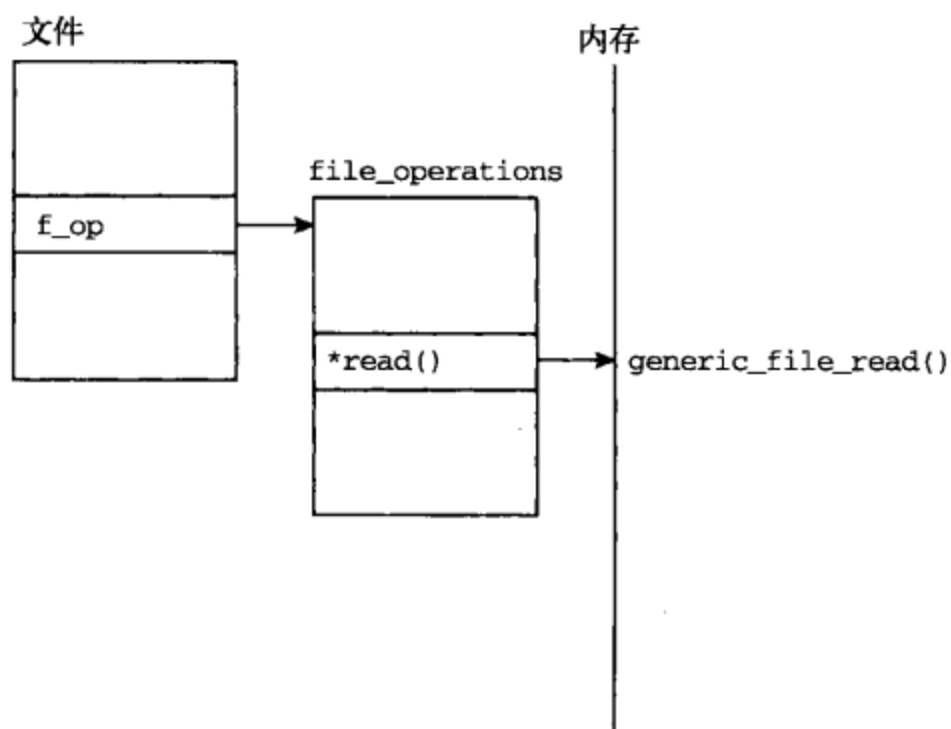


图 6-17 文件操作

当创建或安装文件时，特定文件系统层初始化其文件操作结构。因为我们正在操作的是 ext2 文件系统上的文件，所以，其文件操作结构如下所示：

```

-----
fs/ext2/file.c
42 struct file_operations ext2_file_operations = {
43     .llseek    = generic_file_llseek,
44     .read      = generic_file_read,
45     .write     = generic_file_write,
46     .aio_read  = generic_file_aio_read,
47     .aio_write = generic_file_aio_write,
48     .ioctl     = ext2_ioctl,
49     .mmap      = generic_file_mmap,
50     .open      = generic_file_open,
51     .release   = ext2_release_file,
52     .fsync     = ext2_sync_file,
53     .readv     = generic_file_readv,
54     .writev    = generic_file_writev,
55     .sendfile  = generic_file_sendfile,
56 };
-----
  
```

可以看到，几乎对每一个文件操作，ext2 文件系统都决定采用 Linux 的默认值。那么，文件系统应该在何时实现它自己的文件操作呢？当一个文件系统与 UNIX 的文件系统差异很大时，也许有必要给出几个额外的步骤使得 Linux 可以与这个文件系统进行交互。例如，基于 MSDOS 或 FAT 的文件系统必须实现它们自己的写操作，但是可以使用通用的读操作^①。

了解了 ext2 文件系统层如何把控制权传递给通用文件系统层后，再来考察函数 `generic_file_read()`：

① 详情参见 `fs/fat/file.c`。

```

-----
mm/filemap.c
924 ssize_t
925 generic_file_read(struct file *filp, char __user *buf, size_t count, loff_t *ppos)
926 {
927     struct iovec local_iov = { .iov_base = buf, .iov_len = count };
928     struct kiocb kiocb;
929     ssize_t ret;
930
931     init_sync_kiocb(&kiocb, filp);
932     ret = __generic_file_aio_read(&kiocb, &local_iov, 1, ppos);
933     if (-EIOCBQUEUED == ret)
934         ret = wait_on_sync_kiocb(&kiocb);
935     return ret;
936 }
937
938 EXPORT_SYMBOL(generic_file_read);
-----

```

第 924~925 行：注意，相同的参数沿着上一层的读函数被传递过来。这些参数有：文件指针 `filp`、指向内存缓冲区的指针 `buf`（文件内容将被读到这个缓冲区中）、读入的字符数 `count`、从文件中读数据的起始位置 `ppos`。

第 927 行：创建 `iovec` 结构，该结构包含用户空间缓冲区的地址和长度，读入的数据被存入这个缓冲区中。

第 928~931 行：用文件指针初始化 `kiocb` 结构（`kiocb` 代表内核 I/O 控制块）。

第 932 行：读操作的大部分工作在通用的异步文件读函数中完成。

异步 I/O 操作

`kiocb` 和 `iovec` 是 Linux 内核中协助异步 I/O 操作的两个数据类型。

当进程希望执行输入输出操作，但并不需要等一会儿就马上得到操作结果时，异步 I/O 是非常合适的。对于高级的 I/O 环境尤其适用，因为你可以允许设备对 I/O 请求（而不是进程）进行排序和调度。

在 Linux 中，I/O 向量（`iovec`）表示内存区的地址范围，它的定义如下：

```

-----
include/linux/uio.h
20 struct iovec
21 {
22     void __user *iov_base; /* BSD uses caddr_t (1003.1g requires
    void *) */
23     __kernel_size_t iov_len; /* Must be size_t (1003.1g) */
24 };
-----

```

它只不过是指向内存区的指针和内存区的长度。

内核 I/O 控制块（`kiocb`）是辅助管理 I/O 向量所需的结构，它帮助管理 I/O 向量如何异步地操作以及何时操作。

`__generic_file_aio_read()` 函数使用 `kiocb` 和 `iovec` 结构直接读取 `page_cache`。

第 933~935 行：发出读请求后一直等待，直到读请求完成并返回读操作的结果为止。

回想 `vfs_read()` 中的 `do_sync_read()` 这条线；它最终会通过另一条线调用这一相同的函数。接下来通过考察 `__generic_file_aio_read()` 继续对文件 I/O 进行追踪：

```
-----
mm/filemap.c
835 ssize_t
836 __generic_file_aio_read(struct kiocb *iocb,
const struct iovec *iov,
837     unsigned long nr_segs, loff_t *ppos)
838 {
839     struct file *filp = iocb->ki_filp;
840     ssize_t retval;
841     unsigned long seg;
842     size_t count;
843
844     count = 0;
845     for (seg = 0; seg < nr_segs; seg++) {
846         const struct iovec *iv = &iov[seg];
847         ...
852         count += iv->iov_len;
853         if (unlikely((ssize_t)(count+iv->iov_len) <
0))
854             return -EINVAL;
855         if (access_ok(VERIFY_WRITE, iv->iov_base,
iv->iov_len))
856             continue;
857         if (seg == 0)
858             return -EFAULT;
859         nr_segs = seg;
860         count -= iv->iov_len;
861         break;
862     }
863     ...
-----
```

第 835~842 行：回想一下，`nr_segs` 通过我们的调用程序被设置为 1，并且 `iocb` 和 `iov` 包含了文件指针和缓冲区的信息。现在马上从 `iocb` 中提取文件指针。

第 845~862 行：这个 `for` 循环检验传入的 `iovec` 结构体是否由有效的片段组成。回忆一下，它包含了用户空间的缓冲区信息。

```
-----
mm/filemap.c
...
863
864 /* coalesce the iovecs and go direct-to-BIO for O_DIRECT */
865 if (filp->f_flags & O_DIRECT) {
866     loff_t pos = *ppos, size;
867     struct address_space *mapping;
868     struct inode *inode;
869
870     mapping = filp->f_mapping;
```

```

871     inode = mapping->host;
872     retval = 0;
873     if (!count)
874         goto out; /* skip atime */
875     size = i_size_read(inode);
876     if (pos < size) {
877         retval = generic_file_direct_IO(READ, iocb,
878             iov, pos, nr_segs);
879         if (retval >= 0 && !is_sync_kiocb(iocb))
880             retval = -EIOCBQUEUED;
881         if (retval > 0)
882             *ppos = pos + retval;
883     }
884     file_accessed(filp);
885     goto out;
886 }
...

```

第 863~886 行：仅当读操作是直接 I/O 时才进入这个代码区。直接 I/O 能够绕过页缓存，是某些块设备非常有用的特性。然而，根据我们的需求，根本不会进入这段代码。大多数文件 I/O 把我们的访问路径视为页缓存（这一点将在稍后讨论），它比底层的块设备要快得多。

```

-----
mm/filemap.c
...
887
888     retval = 0;
889     if (count) {
890         for (seg = 0; seg < nr_segs; seg++) {
891             read_descriptor_t desc;
892
893             desc.written = 0;
894             desc.buf = iov[seg].iov_base;
895             desc.count = iov[seg].iov_len;
896             if (desc.count == 0)
897                 continue;
898             desc.error = 0;
899             do_generic_file_read(filp, ppos, &desc, file_read_actor);
900             retval += desc.written;
901             if (!retval) {
902                 retval = desc.error;
903                 break;
904             }
905         }
906     }
907 out:
908     return retval;
909 }
-----

```

第 889~890 行：因为我们的 `iovec` 是有效的，并且只有一个片段，因此只执行一次这个 `for` 循环。

第 891~898 行：将 `iovec` 结构转换成 `read_descriptor_t` 结构。`read_descriptor_t` 结构记录读的状态。此处是对 `read_descriptor_t` 结构的描述：

```
-----
include/linux/fs.h
837  typedef struct {
838      size_t written;
839      size_t count;
840      char __user * buf;
841      int error;
842  } read_descriptor_t;
-----
```

第 838 行：字段 `written` 存放不断变化着的已传送字节数。

第 839 行：字段 `count` 存放不断变化着的未传送字节数。

第 840 行：字段 `buf` 存放缓冲区中的当前位置。

第 841 行：字段 `error` 存放读操作期间遇到的任何错误码。

第 899 行：把新的 `read_descriptor_t` 结构 **desc** 连同文件指针 `filp` 和位置 `ppos` 一起传给 `do_generic_file_read()`。`file_read_actor()`^①是一个函数，它把一个页面复制到由 `desc` 指定的用户空间缓冲区。

第 900~909 行：计算已读字节数，并把它返回给调用者。

这里进入了 `read()` 的内部，我们准备访问页缓存^②，然后确定我们想要读的文件区是否已经在 RAM 中，如果在，就不必直接访问块设备。

2. 追踪页缓存

回想一下我们遇到的最后一个函数，它给 `do_generic_file_read()` 传递了文件指针 `filp`、偏移量 `ppos`、`read_descriptor_t` 类型的 `desc`，以及函数 `file_read_actor` 这些参数。

```
-----
include/linux/fs.h
1420 static inline void do_generic_file_read(struct file * filp, loff_t *ppos,
1421      read_descriptor_t * desc,
1422      read_actor_t actor)
1423 {
1424     do_generic_mapping_read(filp->f_mapping,
1425         &filp->f_ra,
1426         filp,
1427         ppos,
1428         desc,
1429         actor);
1430 }
-----
```

第 1420~1430 行：`do_generic_file_read()` 只不过是对 `do_generic_mapping_read()` 的封装。`filp->f_mapping` 是指向 `address_space` 对象的指针，`filp->f_ra` 是一个存放文件预读

① `file_read_actor()` 参见 `mm/filemap.c` 的第 794 行。

② 6.4 节中描述了页缓存。

状态地址的结构^①。

因此，我们可以通过文件指针中的 `address_space` 对象，把对文件的读转换成对页缓存的读。由于 `do_generic_mapping_read()` 是一个非常长的函数，带有许多分支情况，因此，我们设法尽可能简单地对这些代码进行分析。

```
-----
mm/filemap.c
645 void do_generic_mapping_read(struct address_space *mapping,
646     struct file_ra_state *_ra,
647     struct file * filp,
648     loff_t *ppos,
649     read_descriptor_t * desc,
650     read_actor_t actor)
651 {
652     struct inode *inode = mapping->host;
653     unsigned long index, offset;
654     struct page *cached_page;
655     int error;
656     struct file_ra_state ra = *_ra;
657
658     cached_page = NULL;
659     index = *ppos >> PAGE_CACHE_SHIFT;
660     offset = *ppos & ~PAGE_CACHE_MASK;
-----
```

第 652 行：从 `address_space` 提取正在读的文件的索引节点。

第 658~660 行：把 `cached_page` 初始化成 `NULL`，直到可以确定它是否在页缓存中。同时计算基于页缓存约束的 `index` 和 `offset`。`index` 对应页缓存中的页号，而 `offset` 对应页内偏移。当页大小是 4 096 字节时，对文件指针右移 12 位就得到了页的索引。

“页缓存可以在多于一页的大块内完成，因为这样可获得更高的吞吐量” (`linux/pagemap.h`)。`PAGE_CACHE_SHIFT` 和 `PAGE_CACHE_MASK` 是控制页缓存的结构和页缓存大小的宏：

```
-----
mm/filemap.c
661
662     for (;;) {
663         struct page *page;
664         unsigned long end_index, nr, ret;
665         loff_t isize = i_size_read(inode);
666
667         end_index = isize >> PAGE_CACHE_SHIFT;
668
669         if (index > end_index)
670             break;
671         nr = PAGE_CACHE_SIZE;
672         if (index == end_index) {
-----
```

^① 关于这个字段以及预读优化的更多信息参见“文件结构”部分。

```

673     nr = isize & ~PAGE_CACHE_MASK;
674     if (nr <= offset)
675         break;
676 }
677
678 cond_resched();
679 page_cache_readahead(mapping, &ra, filp, index);
680
681 nr = nr - offset;
-----

```

第 662~681 行：这段代码在页缓存上反复循环，获取足够的页以得到读命令请求的字节。

```

-----
mm/filemap.c
682 find_page:
683     page = find_get_page(mapping, index);
684     if (unlikely(page == NULL)) {
685         handle_ra_miss(mapping, &ra, index);
686         goto no_cached_page;
687     }
688     if (!PageUptodate(page))
689         goto page_not_up_to_date;
-----

```

第 682~689 行：我们试图找到第一个被请求的页。如果这个页不在页缓存中，就跳转到标号 `no_cached_page` 处。如果该页不是最新的，就跳转到标号 `page_not_up_to_date` 处。`find_get_page()` 使用地址空间的基树来查找索引为 `index` 的页，其中 `index` 是指定的偏移量。

```

-----
mm/filemap.c
690 page_ok:
691     /* If users can be writing to this page using arbitrary
692      * virtual addresses, take care about potential aliasing
693      * before reading the page on the kernel side.
694      */
695     if (mapping_writably_mapped(mapping))
696         flush_dcache_page(page);
697
698     /*
699      * Mark the page accessed if we read the beginning.
700      */
701     if (!offset)
702         mark_page_accessed(page);
703     ...
714     ret = actor(desc, page, offset, nr);
715     offset += ret;
716     index += offset >> PAGE_CACHE_SHIFT;
717     offset &= ~PAGE_CACHE_MASK;
718
719     page_cache_release(page);
720     if (ret == nr && desc->count)

```

```

721     continue;
722     break;
723

```

第 690~723 行：这些内嵌的注释已给出详细描述，因此没有必要赘述。请注意第 656 行~658 行，如果要获取更多的页，就立刻返回到循环的开始，在那里，第 714~716 行对 `index` 和 `offset` 的处理有助于选择下一个要获取的页。如果没有更多的页要读，就跳出 `for` 循环。

```

-----
mm/filemap.c
724 page_not_up_to_date:
725     /* Get exclusive access to the page ... */
726     lock_page(page);
727
728     /* Did it get unhashed before we got the lock? */
729     if (!page->mapping) {
730         unlock_page(page);
731         page_cache_release(page);
732         continue;
733
734
735     /* Did somebody else fill it already? */
736     if (PageUptodate(page)) {
737         unlock_page(page);
738         goto page_ok;
739     }
740

```

第 724~740 行：如果该页不是最新的，就再检查一次；如果该页现在是最新的，就立刻返回到标号 `page_ok` 处。否则，将设法获取对该页的独占访问；这将导致我们睡眠，直到获得对该页的独占访问。获得独占访问后，就来看看这个页是否企图从页缓存中删除自己，如果是，在返回到 `for` 循环顶部前赶紧继续向前。如果它仍然存在并且现在是最新的，就对页解锁并跳转到标号 `page_ok` 处。

```

-----
mm/filemap.c
741 readpage:
742     /* ... and start the actual read. The read will unlock the page. */
743     error = mapping->a_ops->readpage(filp, page);
744
745     if (!error) {
746         if (PageUptodate(page))
747             goto page_ok;
748         wait_on_page_locked(page);
749         if (PageUptodate(page))
750             goto page_ok;
751         error = -EIO;
752     }
753

```

```

754     /* UHHUH! A synchronous read error occurred. Report it */
755     desc->error = error;
756     page_cache_release(page);
757     break;
758
-----

```

第 741~743 行：如果该页不是最新的，就跳到前一个标号 `page_not_up_to_date` 处，并对该页加锁。实际的读操作 `mapping->a_ops->readpage(filp, page)` 对该页进行解锁（我们将会稍微深入地对 `readpage()` 进行追踪，但是让我们首先完成当前的解释）。

第 746~750 行：如果成功地读取了一个页，检查它是否是最新的，如果是则跳转到标号 `page_ok` 处。

第 751~758 行：如果发生同步读错误，就在 `desc` 中记录这个错误，同时从页缓存释放该页，并跳出 `for` 循环。

```

-----
mm/filemap.c
759 no_cached_page:
760     /*
761     * Ok, it wasn't cached, so we need to create a new
762     * page..
763     */
764     if (!cached_page) {
765         cached_page = page_cache_alloc_cold(mapping);
766         if (!cached_page) {
767             desc->error = -ENOMEM;
768             break;
769         }
770     }
771     error = add_to_page_cache_lru(cached_page, mapping,
772                                 index, GFP_KERNEL);
773     if (error) {
774         if (error == -EEXIST)
775             goto find_page;
776         desc->error = error;
777         break;
778     }
779     page = cached_page;
780     cached_page = NULL;
781     goto readpage;
782 }
-----

```

第 768~772 行：如果要读的页没有被缓存，就在地址空间中分配一个新的页，并把它添加到最近最少使用（LRU）缓存和页缓存中。

第 773~775 行：向缓存添加页时，如果因为页已经存在而产生错误，就跳转到标号 `find_page` 处并再试一次。如果多个进程试图读取同一个未缓冲的页，也可能会发生这种情况；一个进程试图分配页并且分配成功的话，另一个进程试图再分配时就会发现该页已经存在。

第 776~777 行：如果向缓存添加页时出现的错误并不是页已存在的错误，则记录该错误并跳

出 for 循环。

第 779~781 行：当我们成功地分配了页并将页加入页缓存和 LRU 缓存后，就让指针 page 指向新页，并通过跳转到标号 readpage 处准备开始读取该页。

```
-----
mm/filemap.c
784  *_ra = ra;
785
786  *ppos = ((loff_t) index << PAGE_CACHE_SHIFT) + offset;
787  if (cached_page)
788      page_cache_release(cached_page);
789  file_accessed(filp);
790 }
```

第 786 行：计算基于页缓存索引和偏移量的实际偏移量。

第 787~788 行：如果分配了一个新页，并且能够把它正确地添加到页缓存中的话，就删除这个页。

第 789 行：通过索引节点更新文件的最后一次访问时间。

这个函数中描述的逻辑是页缓存的核心。注意，页缓存如何避免触及任何特定文件系统的数据。这使得 Linux 内核不用考虑底层文件系统的结构，用页缓存就可以缓存各种各样的页。因此，页缓存能够同时容纳来自 MINIX、ext2 以及 MSDOS 的页。

通过调用地址空间的 readpage() 函数，页缓存维护着特定文件系统层之间的差异。每个特定的文件系统都需要实现自己的 readpage()。因此，当通用文件系统层调用 mapping->a_ops->readpage() 时，它从文件系统驱动程序的 address_space_operations 结构中调用具体的 readpage() 函数，对于 ext2 文件系统而言，readpage() 定义如下：

```
-----
fs/ext2/inode.c
676 struct address_space_operations ext2_aops = {
677     .readpage      = ext2_readpage,
678     .readpages     = ext2_readpages,
679     .writepage     = ext2_writepage,
680     .sync_page     = block_sync_page,
681     .prepare_write  = ext2_prepare_write,
682     .commit_write   = generic_commit_write,
683     .bmap          = ext2_bmap,
684     .direct_IO      = ext2_direct_IO,
685     .writepages     = ext2_writepages,
686 };
```

因此，readpage() 实际上调用了 ext2_readpage()：

```
-----
fs/ext2/inode.c
616 static int ext2_readpage(struct file *file, struct page *page)
617 {
```

```

618     return mpage_readpage(page, ext2_get_block);
619 }

```

ext2_readpage()调用通用文件系统层的 mpage_readpage(),但会给后者传入特定文件系统层的函数 ext2_get_block()。

通用文件系统函数 mpage_readpage()把 get_block()函数作为它的第二个参数。每个文件系统都实现某些 I/O 函数,这些函数专门对应于这种文件系统的格式;get_block()便是这些函数中的一个。文件系统的 get_block()函数将 address_space 页中的逻辑块映射为特定文件系统布局中实际的设备块。让我们来分析 mpage_readpage()的细节:

```

-----
fs/mpage.c
358 int mpage_readpage(struct page *page, get_block_t get_block)
359 {
360     struct bio *bio = NULL;
361     sector_t last_block_in_bio = 0;
362
363     bio = do_mpage_readpage(bio, page, 1,
364         &last_block_in_bio, get_block);
365     if (bio)
366         mpage_bio_submit(READ, bio);
367     return 0;
368 }
-----

```

第 360~361 行:为了管理地址空间所使用的 bio 结构,我们为其分配空间以管理正试图从设备读取的页。

第 363~364 行:调用 do_mpage_readpage()把逻辑页转换成由实际的页和块组成的 bio 结构。bio 结构记录着与块 I/O 相关的信息。

第 365~367 行:将新创建的 bio 结构发送到 mpage_bio_submit()并返回。

让我们花点时间,从宏观上扼要地阐述一下迄今为止 read 函数的流程。

- ❑ 利用来自 open()调用的文件描述符定位文件指针,并从文件指针获得索引节点。
- ❑ 文件系统层在内存的页缓存中检查与给定索引节点对应的一个或多个页。
- ❑ 如果在页缓存中找不到所需的页,文件系统层使用特定文件系统的驱动程序把所请求的文件片断转换成特定设备上的 I/O 块。
- ❑ 在页缓存的 address_space 中为页分配空间,并建立一个 bio 结构,该结构把新分配的页与块设备上的扇区对应起来。

mpage_readpage()是建立 bio 结构的函数,它把从页缓存中新分配的页和 bio 结构联系起来。不过,此时页中还没有数据。因此,文件系统层需要块设备的驱动程序来完成到设备的实际接口。这是由 mpage_bio_submit()中的 submit_bio()函数来完成的:

```

-----
fs/mpage.c
90 struct bio *mpage_bio_submit(int rw, struct bio *bio)
91 {

```



```

92  bio->bi_end_io = mpage_end_io_read;
93  if (rw == WRITE)
94      bio->bi_end_io = mpage_end_io_write;
95  submit_bio(rw, bio);
96  return NULL;
97  }

```

第 90 行：首先要注意的是：通过传递 `rw` 参数，`mpage_bio_submit()` 既可以为 `read` 调用服务，又可以为 `write` 调用服务。它提交一个 `bio` 结构，在 `read` 调用的情况下，这个结构是空的，所以必须被填充。在 `write` 调用的情况下，`bio` 结构是已经填充好的，块设备驱动程序把其中的内容复制到自己的设备中。

第 92~94 行：如果正在读或正在写页面，就设置合适的函数作为 I/O 结束时调用的函数。

第 95~96 行：调用 `submit_bio()` 并返回 `NULL`。回想一下，`mpage_readpage()` 对 `mpage_bio_submit()` 的返回值不做任何处理。

`submit_bio()` 是 Linux 内核通用块设备驱动层的组成部分。

```

-----
drivers/block/ll_rw_blk.c
2433 void submit_bio(int rw, struct bio *bio)
2434 {
2435     int count = bio_sectors(bio);
2436
2437     BIO_BUG_ON(!bio->bi_size);
2438     BIO_BUG_ON(!bio->bi_io_vec);
2439     bio->bi_rw = rw;
2440     if (rw & WRITE)
2441         mod_page_state(pgpout, count);
2442     else
2443         mod_page_state(pgpgin, count);
2444
2445     if (unlikely(block_dump)) {
2446         char b[BDEVNAME_SIZE];
2447         printk(KERN_DEBUG "%s(%d): %s block %Lu on %s\n",
2448             current->comm, current->pid,
2449             (rw & WRITE) ? "WRITE" : "READ",
2450             (unsigned long long)bio->bi_sector,
2451             bdevname(bio->bi_bdev, b));
2452     }
2453
2454     generic_make_request(bio);
2455 }

```

第 2433~2443 行：这几个调用启用一些调试：设置 `bio` 结构的读/写属性，执行部分页状态的管理工作。

第 2445~2452 行：这几行处理偶然发生块转储时的情况。它将抛出一个调试消息。

第 2454 行：`generic_make_request()` 包含主要功能，并利用特定块设备驱动程序的请求队列来处理块的 I/O 操作。

下面是 generic_make_request() 部分内嵌的注释：

```
-----
drivers/block/ll_rw_blk.c
2336 * The caller of generic_make_request must make sure that bi_io_vec
2337 * are set to describe the memory buffer, and that bi_dev and bi_sector
are
2338 * set to describe the device address, and the
2339 * bi_end_io and optionally bi_private are set to describe how
2340 * completion notification should be signaled.
-----
```

在以上这些阶段，我们构造了 bio 结构，因此，bio_vec 结构被映射到第 2337 行提到的内存缓冲区，bio 结构也用设备地址参数来初始化。如果想要深入到块设备的驱动程序继续研究读操作，请参考 5.2.1 节，它描述了块设备的驱动程序如何处理请求队列以及设备的具体硬件约束条件。图 6-18 说明了 read() 系统调用是如何穿越内核的各个功能层的。

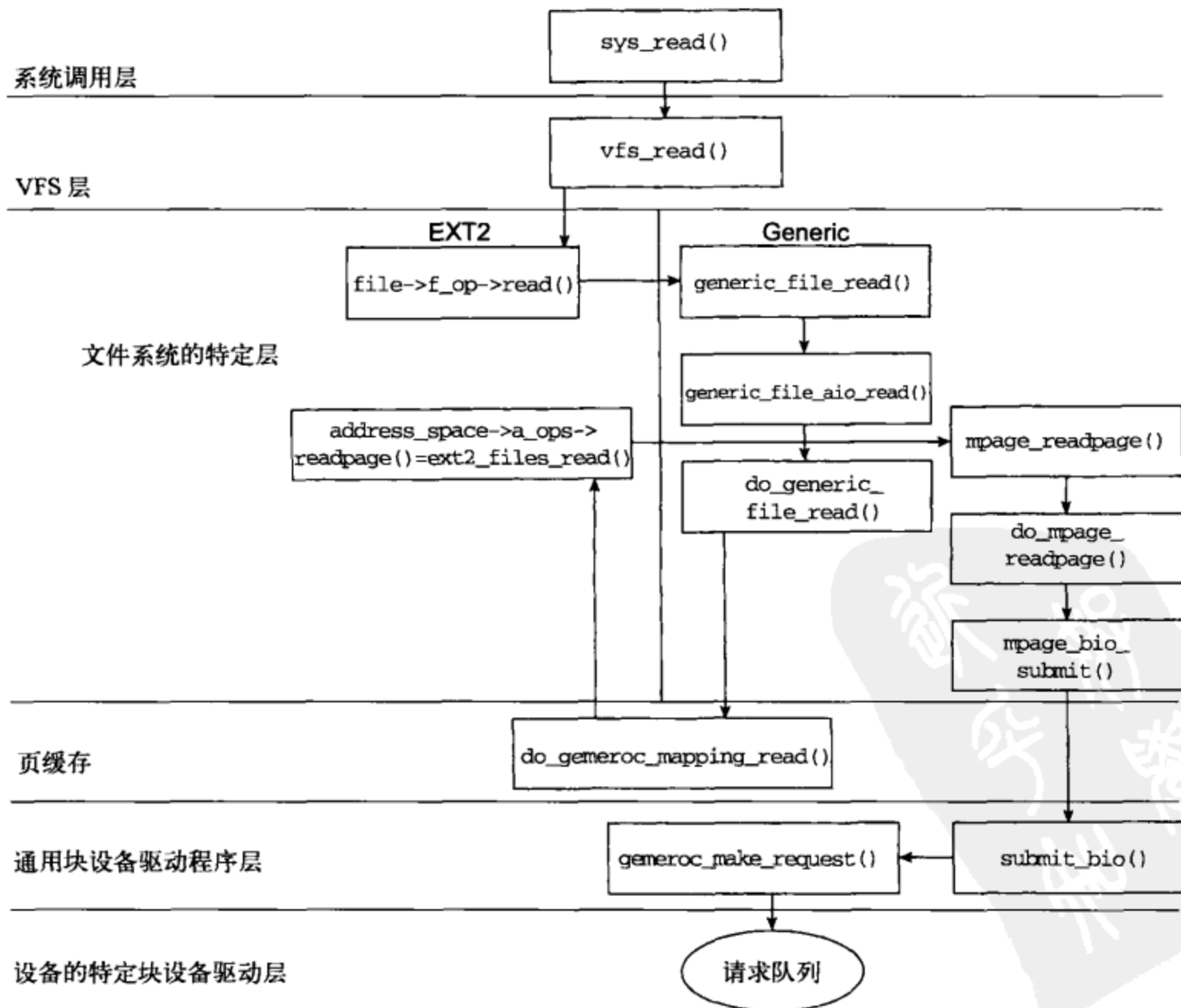


图 6-18 read() 自顶而下的遍历

块设备的驱动程序读取实际的数据并把它们放入 bio 结构以后，我们追踪的代码就展开了。页缓存中新分配的页被填充，页的引用被传回 VFS 层，并把页中的数据复制到指定的用户空间区域——这就是最初由 read() 调用给出的缓冲区。

也许读者会问：“难道只介绍一半的内容吗？如果我们想要写，而不是读呢？”

我们希望以上描述给出一条清晰的思路，read() 调用在整个 Linux 内核中穿越的轨迹与 write() 调用类似，但还是要概述一下二者的差异。

6.5.4 write()

write() 调用被映射到 sys_write()，而 sys_write() 被映射 vfs_write()，与 read() 调用的映射方式完全相同：

```
-----
fs/read_write.c
244 ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)
245 {
...
259     ret = file->f_op->write(file, buf, count, pos);
...
268 }
-----
```

vfs_write() 使用 file_operations 的通用 write 函数，以确定使用哪一个具体文件系统层的 write。在我们举的例子 ext2 中，可以通过 ext2_file_operations 结构进行转换：

```
-----
fs/ext2/file.c
42 struct file_operations ext2_file_operations = {
43     .llseek    = generic_file_llseek,
44     .read      = generic_file_read,
45     .write     = generic_file_write,
...
56 };
-----
```

第 44~45 行：此处调用 generic_file_write()，而不是 generic_file_read()。

generic_file_write() 获得文件的锁，以防止两个写程序同时写一个文件，并调用 generic_file_write_nolock()。后者把文件指针和缓冲区分别转换成 kiocb 参数和 iovec 参数，最后调用页缓存的写函数 generic_file_aio_write_nolock()。

此处是写和读分道扬镳的地方。如果要写的页不在页缓存中，对设备本身而言，写操作并不是以失败告终。相反，它把页读入页缓存，然后再执行写操作。页缓存中的页不会被立刻写到磁盘上，而是它们被标记为“脏”，所有的脏页都要被定期地写回磁盘。

这是与 read() 函数的 readpage() 类似的函数。在 generic_file_aio_write_nolock() 内部，通过 address_space_operations 的指针访问 prepare_write() 和 commit_write()，这两个函数是文件所驻留的文件系统类型所特有的。回想一下，ext2_aops，我们会看到 ext2 的驱动程序使用它自己的函数 ext2_prepare_write() 和通用函数 generic_commit_write()。

```

-----
fs/ext2/inode.c
628 static int
629 ext2_prepare_write(struct file *file, struct page *page,
630     unsigned from, unsigned to)
631 {
632     return block_prepare_write(page, from, to, ext2_get_block);
633 }
-----

```

第 632 行: `ext2_prepare_write` 只不过是对通用文件系统函数 `block_prepare_write()` 的简单封装, 它将传入 `ext2` 文件系统所特有的函数 `get_block()`。

`block_prepare_write()` 分配写操作所需的新缓冲区。例如, 如果要把数据添加到文件末尾, 就要创建足够的缓冲区, 并把它们和页关联起来以存放新数据。

`generic_commit_write()` 获得给定的页, 并扫描页内的缓冲区, 标记每个脏的页。把写操作分解成准备和提交两步以防止把不完整的写数据从页缓存刷新到块设备。

刷出脏页

`write()` 把已经写过的所有页插入页缓存并标记为脏之后返回。Linux 有一个后台程序 `pdflush`, 它能够在以下两种情况下把脏页从页缓存写到块设备。

- 系统的空闲内存低于阈值。刷出页缓存中的页以释放内存。
- 脏页达到了某个年龄。把某段时间后仍没有写回磁盘的页写到它们的块设备中。

当准备好向磁盘写页时, 后台程序 `pdflush` 将调用特定于文件系统的 `writpages()` 函数。因此, 在我们的例子中, 回想一下 `ext2_file_operation` 结构, 在该结构中, `writpages()` 就等同于 `ext2_writpages()`^①。

```

-----
670 static int
671 ext2_writpages(struct address_space *mapping, struct writeback_control *wbc)
672 {
673     return mpage_writpages(mapping, wbc, ext2_get_block);
674 }
-----

```

与其他通用文件系统函数的具体实现类似, `ext2_writpages()` 只不过是调用了通用文件系统的函数 `mpage_writpages()`, 并把特定于文件系统的函数 `ext2_get_block()` 作为第二参数传递给它。

`mpage_writpages()` 扫描脏页, 并对每个脏页调用 `mpage_writpage()`。与 `mpage_readpage()` 类似, `mpage_writpage()` 返回一个 `bio` 结构, 该结构把页的物理设备布局映射成其物理内存布局。`mpage_writpages()` 接着调用 `submit_bio()` 向块设备的驱动程序发送新创建的 `bio` 结构, 从而把数据传送到设备本身。

① `Pdflush` 后台例程是相当棘手的, 我们的目的是追踪 `write`, 因此忽略其复杂性。然而, 如果你对细节感兴趣的话, 文件 `mm/pdflush.c`、`mm/fs-writeback.c` 和 `mm/page-writeback.c` 中包含了相关代码。

6.6 小结

本章首先分析了通用文件模型所涉及的结构和全局变量，这些结构包括超级块、索引节点、目录项以及文件结构。然后研究了 VFS 的相关结构，并讨论了 VFS 支持各种文件系统的工作机理。

接着，研究了与 VFS 相关的系统调用 `open` 和 `close`，分析了它们之间如何协同工作。然后，追踪了 `read()` 和 `write()` 这两个用户空间的调用——从 VFS 穿越到底层，涵盖了通用文件系统层和特定文件系统层之间的细节。为了对特定文件系统层加以说明，我们以 `ext2` 文件系统的驱动程序为例，分析了内核是如何穿插调用特定于文件系统的驱动程序函数和通用文件系统的函数。这使得我们讨论了页缓存，页缓存是一片内存区域，它存放的最近访问的页是从连接到系统的块设备读取的。

6.7 习题

1. 在什么情况下，你会使用索引节点的 `i_hash` 字段而不是 `i_list` 字段？为什么同一个结构中既有散列表又有线性表？
2. 在我们遇到的所有文件结构中，指出在硬盘上有相应数据结构的结构。
3. 目录项对象用于什么类型的操作？为什么不只用索引节点？
4. 文件描述符和文件结构之间有什么联系？是一对一，多对一，还是一对多？
5. `fd_set` 结构的用途是什么？
6. 什么类型的数据结构保证了页缓存以最大的速度操作？
7. 假如你正在写一个新的文件系统驱动程序。你要用新驱动程序 (`media_fs`) (可以优化多媒体的文件 I/O) 替换 `ext2` 文件系统驱动程序。你会在哪里对 Linux 内核进行修改，以确保用的是你的新驱动程序，而不是 `ext2` 的驱动程序？
8. 页怎样变脏？如何把脏页写回磁盘？



进程调度和内核同步

本章内容

- Linux 的调度程序
- 内核抢占
- 自旋锁和信号量
- 系统时钟：关于时间和定时器

Linux 内核是多任务内核，这意味着多个进程可以同时运行，就好像它们是系统中唯一的进程一样。操作系统在特定时刻选择哪一个进程访问系统的 CPU 是由调度程序来决定的。

调度程序负责在不同的进程之间切换 CPU，并决定进程访问 CPU 的次序。与大多数操作系统一样，Linux 通过时钟中断来触发调度程序。当时钟中断发生时，内核必须决定是否为其他非当前进程让出 CPU，如果能让出的话，接下来应该由哪一个进程获得 CPU。相邻两个时钟中断之间的时间间隔称为时间片（timeslice）。

系统中的进程往往分成两种类型：交互式的和非交互式的。交互式进程非常依赖于 I/O，因此，它们通常用不尽自己的时间片，而是将 CPU 让给其他进程。

非交互式进程则非常依赖于 CPU，它们通常都会花掉大部分（如果不是全部的话）时间片。调度程序必须平衡这两类进程的需求，努力确保每个进程都能获得足够多的时间去完成任务，而不会对其他进程的执行产生不利的影响。

与某些调度程序相似，Linux 还要区别对待另一类进程：实时进程。实时进程必须实时地执行。Linux 支持实时进程，但是它们是处于调度逻辑之外的。简单地说，Linux 的调度程序认为，标记为实时的所有进程的优先级比其他任何进程都高。开发者必须确保实时进程不会贪婪地占有 CPU，并确保它们最终会放弃 CPU。

调度程序通常使用某种类型的进程队列来管理系统中进程的执行。在 Linux 中，这个进程队列称为运行队列^①。在第 3 章中完整地描述了运行队列，但是由于运行队列与调度程序关系密切，这里再次简要阐述一下它的基本原理。

在 Linux 中，运行队列由两个优先级数组组成。

- 活跃数组：存放时间片还没有耗尽的进程。

^① 3.6 节介绍了运行队列。

□ 到期数组：存放时间片已经耗尽的进程。

总的来说，Linux 中调度程序的工作就是：找出优先级最高的活跃进程，允许它们使用 CPU 并投入运行；当进程耗尽自己的时间片时把它放入到期数组。理解了这个总的框架后，再来详细分析 Linux 的调度程序究竟是如何工作的。

7.1 Linux 的调度程序

Linux 2.6 内核引入了一个全新的调度程序，一般称为 $O(1)$ 调度程序，它能够在恒定的时间内完成进程的调度^①。第 3 章讨论了调度程序的基本结构，以及如何初始化新创建的进程，使之参与调度。本节首先描述进程如何在一个单 CPU 系统上执行，当然也会提及多 CPU (SMP) 系统中的调度代码。但是，同一个调度过程通常适用于不同的 CPU。接下来描述调度程序如何通过上下文的切换来换出当前正在运行的进程，最后将简单介绍 2.6 内核中的另一个重大变化：内核抢占。

总的来说，调度程序只不过是对给定数据结构进行操作的一组函数。实现调度程序的所有代码几乎都能在文件 `kernel/sched.c` 和 `include/linux/sched.h` 中找到。要再次强调的一点是：调度程序的代码交替地使用术语“任务”和“进程”。有时候，代码注释中也用“线程”来指代任务或进程。在调度程序中，任务（或进程）是数据结构和控制流的集合。调度程序的代码也引用 `task_struct` 这个 Linux 内核用来记录进程信息的数据结构^②。

7.1.1 选择下一个进程

进程被初始化并放到运行队列后，它应该在某个时刻获得 CPU 的控制权并投入运行。函数 `schedule()` 和 `scheduler_tick()` 负责把 CPU 的控制权传递给不同的进程。`scheduler_tick()` 是一个由内核周期性调用的系统定时器，它把进程标记为需要重新调度。当定时器事件发生时，当前进程就被暂停，Linux 内核本身接管对 CPU 的控制权。待定时器事件结束后，Linux 内核通常把控制权交回刚刚被暂停的进程。然而，当这个进程被标记为需要重新调度时，内核调用 `schedule()` 来选择将要激活哪个进程，当然，并不是在内核接管 CPU 控制权之前正在执行的那个进程。我们把在内核接管控制权之前正在执行的进程称为当前进程（current process）。在某些稍微复杂一些的情况中，内核可以从内核本身获得控制权，这称为内核抢占（kernel preemption）。为简单起见，在接下来的几节中，假定调度程序只需要决定究竟选择用户空间两个进程中的哪一个来获得 CPU 的控制权。

图 7-1 说明了随着时间的推移，如何在各个进程之间传递 CPU 的控制权。如图所示，进程 A 拥有对 CPU 的控制权并且正在执行。系统定时器 `scheduler_tick()` 执行后，将从 A 获得对 CPU 的控制权，并把 A 标记为需要重新调度。Linux 内核调用 `schedule()`，调度程序选择了进程 B，这样，CPU 的控制权就交给了进程 B。

① $O(1)$ 是大 O 表示法，它意味着调度程序的开销是恒定的，与系统当前的负载无关。

② 第 3 章详细介绍了 `task_struct` 结构。

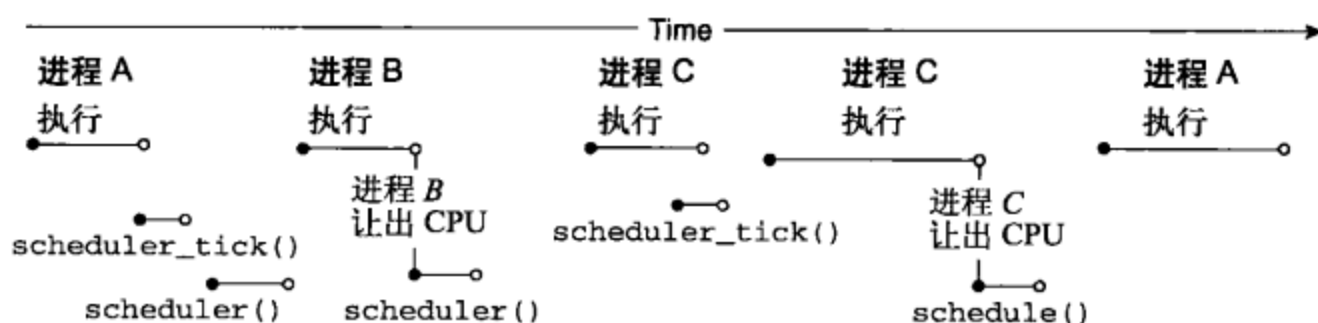


图 7-1 调度进程

进程 B 执行了一会儿后主动让出了 CPU。这种情况通常发生在进程等待资源的时候。进程 B 调用了 `schedule()`，调度程序选择进程 C 来执行。

进程 C 开始执行，直到 `scheduler_tick()` 发生，`scheduler_tick()` 没有把 C 标记为需要重新调度。显然，`schedule()` 不被调用，因此，C 再次获得 CPU 的控制权。

通过调用 `schedule()`，进程 C 让出了 CPU，`schedule()` 决定进程 A 应该获得 CPU 的控制权，于是 A 再次开始执行。

`schedule()` 决定接下来要执行的是哪个进程，`scheduler_tick()` 决定哪个进程必须让出 CPU。此处，首先分析 `schedule()`，然后分析 `scheduler_tick()`。这两个函数一起完美地展示了调度程序的控制流程。

```
-----
kernel/sched.c
2184 asmlinkage void schedule(void)
2185 {
2186     long *switch_count;
2187     task_t *prev, *next;
2188     runqueue_t *rq;
2189     prio_array_t *array;
2190     struct list_head *queue;
2191     unsigned long long now;
2192     unsigned long run_time;
2193     int idx;
2194
2195     /*
2196     * Test if we are atomic. Since do_exit() needs to call into
2197     * schedule() atomically, we ignore that path for now.
2198     * Otherwise, whine if we are scheduling when we should not be.
2199     */
2200     if (likely(!(current->state & (TASK_DEAD | TASK_ZOMBIE)))) {
2201         if (unlikely(in_atomic())) {
2202             printk(KERN_ERR "bad: scheduling while atomic!\n");
2203             dump_stack();
2204         }
2205     }
2206
2207 need_resched:
2208     preempt_disable();
2209     prev = current;
2210     rq = this_rq();
2211
2212     release_kernel_lock(prev);
```

```

2213  now = sched_clock();
2214  if (likely(now - prev->timestamp < NS_MAX_SLEEP_AVG))
2215      run_time = now - prev->timestamp;
2216  else
2217      run_time = NS_MAX_SLEEP_AVG;
2218
2219  /*
2220   * Tasks with interactive credits get charged less run_time
2221   * at high sleep_avg to delay them losing their interactive
2222   * status
2223   */
2224  if (HIGH_CREDIT(prev))
2225      run_time /= (CURRENT_BONUS(prev) ? : 1);
-----

```

第 2213~2218 行：计算进程被调度程序激活后运行了多长时间。如果进程被激活的时间长度大于最大平均睡眠时间（NS_MAX_SLEEP_AVG），则将其运行时间设置为最大平均睡眠时间。

这就是 Linux 内核在其他代码片段中所谓的时间片。时间片既指调度程序中断的间隔时间，又指进程使用 CPU 的时间长度。如果进程耗尽了时间片，它就到期了，不再是活跃进程。时间戳是一个用来确定进程使用 CPU 的时间长度的绝对值。调度程序利用时间戳来减小已使用过 CPU 的进程的时间片。

例如，假定进程 A 有一个 50 个时钟周期的时间片。它使用了 5 个时钟周期的 CPU，然后为其他进程让出 CPU。内核利用时间戳便可确定进程 A 的时间片还剩 45 个周期。

第 2224~2225 行：交互式进程是那些花费大量时间用于等待输入的进程。键盘控制器就是一个很好的交互式进程的例子，它大部分时间都在等待输入，但是当它有任务要执行时，用户又希望它拥有高优先级。

交互式进程，指那些交互信用超过 100（默认值）的进程，它们的有效 run_time 被 (sleep_avg/max_sleep_avg*MAX_BONUS(10)) 除^①：

```

-----
kernel/sched.c
2226
2227  spin_lock_irq(&rq->lock);
2228
2229  /*
2230   * if entering off of a kernel preemption go straight
2231   * to picking the next task.
2232   */
2233  switch_count = &prev->nivcs;
2234  if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
2235      switch_count = &prev->nvcsw;
2236      if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
2237                  unlikely(signal_pending(prev))))
2238          prev->state = TASK_RUNNING;
2239      else
2240          deactivate_task(prev, rq);
2241  }
-----

```

① BONUS 是对高优先级进程的调度奖励。

第 2227 行：该函数获得即将要被修改的运行队列的锁。

第 2233~2241 行：如果由于内核抢占前一个进程而进入了 `schedule()`，那么，若该进程有挂起的信号，就让前一个进程有继续运行的资格，这就意味着内核紧接着抢占了正常的处理；因此，代码包含了两个 `unlikely()` 语句^①。如果接下来没有继续发生内核抢占的话，就从运行队列删除被抢占的进程并继续选择下一个要运行的进程。

```
-----
kernel/sched.c
2243  cpu = smp_processor_id();
2244  if (unlikely(!rq->nr_running)) {
2245      idle_balance(cpu, rq);
2246      if (!rq->nr_running) {
2247          next = rq->idle;
2248          rq->expired_timestamp = 0;
2249          wake_sleeping_dependent(cpu, rq);
2250          goto switch_tasks;
2251      }
2252  }
2253
2254  array = rq->active;
2255  if (unlikely(!array->nr_active)) {
2256      /*
2257       * Switch the active and expired arrays.
2258       */
2259      rq->active = rq->expired;
2260      rq->expired = array;
2261      array = rq->active;
2262      rq->expired_timestamp = 0;
2263      rq->best_expired_prio = MAX_PRIO;
2264  }
-----
```

第 2243 行：利用 `smp_processor_id()` 来获取当前 CPU 的标识符。

第 2244~2252 行：如果运行队列中没有进程，就将空闲进程设置为下一个要执行的进程，并且把运行队列的到期时间戳重置为 0。在多处理器系统中，在这之前还要先检查在其他 CPU 上是否有本 CPU 可以获取的进程在运行。实际上，在系统中的所有 CPU 上都加载了用于平衡负载的空闲进程。仅当没有进程能够从其他 CPU 移出时，才把空闲设置为运行队列的下一个进程并重置到期的时间戳。

第 2255~2264 行：如果运行队列的活跃数组为空，就交换活跃数组和到期数组的指针，然后选择一个新进程来运行。

```
-----
kernel/sched.c
2266  idx = sched_find_first_bit(array->bitmap);
2267  queue = array->queue + idx;
2268  next = list_entry(queue->next, task_t, run_list);
2269
```

① 关于 `unlikely` 例程的更多信息，请参阅第 2 章。

```

2270  if (dependent_sleeper(cpu, rq, next)) {
2271      next = rq->idle;
2272      goto switch_tasks;
2273  }
2274
2275  if (!rt_task(next) && next->activated > 0) {
2276      unsigned long long delta = now - next->timestamp;
2277
2278      if (next->activated == 1)
2279          delta = delta * (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128;
2280
2281      array = next->array;
2282      dequeue_task(next, array);
2283      recalc_task_prio(next, next->timestamp + delta);
2284      enqueue_task(next, array);
2285  }
next->activated = 0;
-----

```

第 2266~2268 行：调度程序利用 `sched_find_first_bit()` 找到优先级最高的进程来运行，然后，让变量 `queue` 指向优先级数组的某个元素中存放的链表。最后，把变量 `next` 初始化为指向链表 `queue` 中的第一个进程。

第 2270~2273 行：如果将要被激活的进程必需等待正在睡眠的兄弟进程，那么就激活一个新的进程，并且跳转到 `switch_tasks` 处继续执行调度函数。

假设有一个进程 A，它派生出进程 B，用于从设备读取数据，此时进程 A 必须等待进程 B 完成后才能继续执行。如果此时调度程序选择激活进程 A，函数 `dependent_sleeper()` 就会发现进程 A 正在等待进程 B，于是调度程序将选择激活一个新的进程。

第 2275~2285 行：如果进程 `next` 的 `activated` 属性大于 0 并且 `next` 不是实时进程，就从链表 `queue` 中删除 `next` 进程，并在重新计算其优先级后将它再次放入队列 `queue` 中。

第 2286 行：将进程 `next` 的 `activated` 属性设置为 0，随后，`next` 以这个属性值运行。

```

-----
kernel/sched.c
2287 switch_tasks:
2288     prefetch(next);
2289     clear_tsk_need_resched(prev);
2290     RCU_qsctr(task_cpu(prev))++;
2291
2292     prev->sleep_avg -= run_time;
2293     if ((long)prev->sleep_avg <= 0) {
2294         prev->sleep_avg = 0;
2295         if (!(HIGH_CREDIT(prev) || LOW_CREDIT(prev)))
2296             prev->interactive_credit--;
2297     }
2298     prev->timestamp = now;
2299
2300     if (likely(prev != next)) {
2301         next->timestamp = now;
2302         rq->nr_switches++;
2303         rq->curr = next;

```

```

2304     ++*switch_count;
2305
2306     prepare_arch_switch(rq, next);
2307     prev = context_switch(rq, prev, next);
2308     barrier();
2309
2310     finish_task_switch(prev);
2311 } else
2312     spin_unlock_irq(&rq->lock);
2313
2314     reacquire_kernel_lock(current);
2315     preempt_enable_no_resched();
2316     if (test_thread_flag(TIF_NEED_RESCHED))
2317         goto need_resched;
2318 }

```

第 2288 行：试图把进程 `next` 的 `task` 结构的内容预取到 CPU 的 L1 高速缓存（详情可参阅 `include/linux/prefetch.h`）。

第 2290 行：必须通知当前 CPU 目前正在进行上下文切换。这样，具有多 CPU 的设备才能够确保正在被其他 CPU 共享的数据可被独占地访问。这个过程称为 RCU。详情可参阅 <http://lse.sourceforge.net/locking/rcupdate.html>。

第 2292~2298 行：将进程 `prev` 的 `sleep_avg` 属性减去它已经运行的时间，并对负值进行调整。如果该进程既不是交互式的也不是非交互式的，其交互信用就处于低信用和高信用之间，由于进程 `prev` 的平均睡眠时间很短，因此，降低其交互信用，并将其时间戳更新为当前时间。这样可以帮助调度程序了解给定进程已经使用了多少 CPU 时间，继而估算进程以后将要使用多少 CPU 时间。

第 2300~2304 行：如果调度程序没有选择同一个进程，就设置新进程的时间戳，增加运行队列的计数器，并把新进程设置为当前进程。

第 2306~2308 行：这几行代码描述了用汇编语言编写的 `context_switch()`。下一节深入讨论上下文切换时，还会谈到这个函数。

第 2314~2318 行：重新获得内核锁，启用抢占，并判断是否需要立刻重新调度；如果需要重新调度，则回到 `schedule()` 的开头。

执行完 `context_switch()` 后，有可能需要重新调度。因为，也许 `scheduler_tick()` 已经把一个新进程标记为需要重新调度了，或者还有一种情况：当启用抢占时，新进程已经被标记为需要重新调度了。我们继续重新调度进程（并对它们进行上下文切换），直到找到一个不需要重新调度的进程为止。离开 `schedule()` 的进程成为将要在这个 CPU 上执行的新进程。

7.1.2 上下文切换

`/kernel/sched.c` 中的 `schedule()` 调用了 `context_switch()`，该函数用来完成与硬件相关的内存环境和处理器状态的切换。一言以蔽之，上下文切换就是指用下一个进程来交换当前进程。函数 `context_switch()` 执行下一个进程并返回指向前一个进程的进程结构的指针。


```

-----
kernel/sched.c
1048 /*
1049 * context_switch - switch to the new MM and the new
1050 * thread's register state.
1051 */
1052 static inline
1053 task_t * context_switch(runqueue_t *rq, task_t *prev, task_t *next)
1054 {
1055     struct mm_struct *mm = next->mm;
1056     struct mm_struct *oldmm = prev->active_mm;
1057     ...
1063     switch_mm(oldmm, mm, next);
1058     ...
1072     switch_to(prev, next, prev);
1073
1074     return prev;
1075 }
-----

```

此处，描述了 context_switch 的两个工作：一个是切换虚拟内存映射；另一个是切换进程/线程的结构。第一个工作是由函数 switch_mm() 完成的，该函数使用了许多与硬件相关的内存管理的结构和寄存器：

```

-----
#include/asm-i386/mmu_context.h
026 static inline void switch_mm(struct mm_struct *prev,
027     struct mm_struct *next,
028     struct task_struct *tsk)
029 {
030     int cpu = smp_processor_id();
031
032     if (likely(prev != next)) {
033         /* stop flush ipis for the previous mm */
034         cpu_clear(cpu, prev->cpu_vm_mask);
035 #ifdef CONFIG_SMP
036         cpu_tlbstate[cpu].state = TLBSTATE_OK;
037         cpu_tlbstate[cpu].active_mm = next;
038 #endif
039         cpu_set(cpu, next->cpu_vm_mask);
040
041         /* Re-load page tables */
042         load_cr3(next->pgd);
043
044         /*
045          * load the LDT, if the LDT is different:
046          */
047         if (unlikely(prev->context.ldt != next->context.ldt))
048             load_LDT_nolock(&next->context, cpu);
049     }
050 #ifdef CONFIG_SMP
051     else {
052         ...
053     }
054 }
-----

```

第 39 行：将新进程绑定到当前处理器。

第 42 行：用来切换内存上下文的代码使用了 x86 的硬件寄存器 cr3，该寄存器中存放了为给定进程进行所有分页操作的基地址。新的页全局描述符即 next->pgd 被加载到 cr3。

第 47 行：大多数进程共享同一个 LDT。如果该进程请求另一个 LDT，则从新进程的 next->context 结构中把这个 LDT 加载到此处。

接着，/kernel/sched.c 中的 context_switch() 的另一半代码调用了宏 switch_to()，这个宏再调用 C 函数 __switch_to()。对于 x86 和 PPC 而言，体系结构独立的部分和体系结构相关的部分，其分界线都是宏 switch_to()。

1. 追踪 x86 中 switch_to() 的踪迹

x86 的代码比 PPC 的代码更为紧凑。下面是为 __switch_to() 编写的与体系结构相关的代码。task_struct（不是 thread_struct）被传递到 __switch_to()。接下来要讨论的代码就是为调用 C 函数 __switch_to()（第 23 行）而设计的内联汇编代码，__switch_to() 以适当的 task_struct 结构作为参数。

context_switch 用于获得三个进程的指针：prev、next 和 last。此外，还有当前进程的指针。

从宏观的角度来看，调用 switch_to() 时将会发生什么？调用 switch_to() 后进程的指针发生了什么变化？

图 7-2 表示分别用进程 A、B、C 对 switch_to() 进行调用。

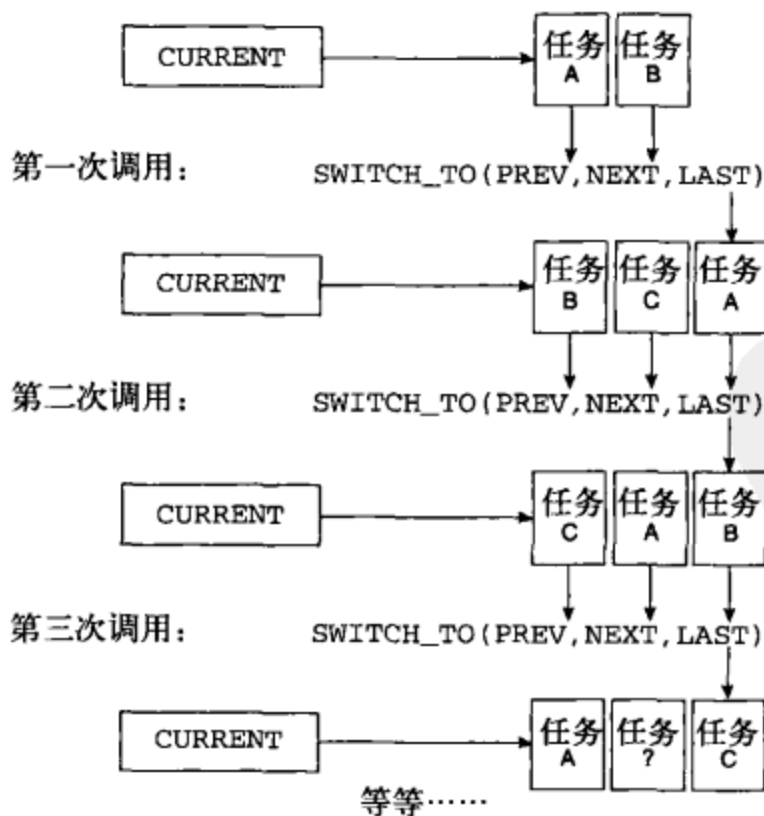


图 7-2 switch_to 调用

我们想切换进程 A 和进程 B。在第一次调用 switch_to() 之前：

- current → A
- prev → A, next → B

第一次调用 `switch_to()` 之后:

□ `current` → B

□ `last` → A

现在, 我们想切换进程 B 和进程 C。在第二次调用 `switch_to()` 之前:

□ `current` → B

□ `prev` → B, `next` → C

第二次调用 `switch_to()` 之后:

□ `current` → C

□ `last` → B

从第二个调用返回后, `current` 指向了进程 C, `last` 指向了进程 B。

进程 A 再一次被切换进来, 这个过程将周而复始地执行下去……

`switch_to()` 函数中的内联汇编是内核中汇编技巧应用的精彩实例, 也是 gcc C 扩展的一个好例子。第 2 章专门描述过这个函数的特色。接下来仔细分析这段代码。

```
-----
#include/asm-i386/system.h
012 extern struct task_struct * FASTCALL(__switch_to(struct task_struct *prev,
struct task_struct *next));

015 #define switch_to(prev,next,last) do {      \
016     unsigned long esi,edi;                  \
017     asm volatile("pushfl\n\t"              \
018         "pushl %%ebp\n\t"                   \
019         "movl %%esp,%0\n\t" /* save ESP */    \
020         "movl %5,%%esp\n\t" /* restore ESP */ \
021         "movl $1f,%1\n\t" /* save EIP */     \
022         "pushl %6\n\t" /* restore EIP */     \
023         "jmp __switch_to\n\t"               \
023         "1:\n\t"                           \
024         "popl %%ebp\n\t"                    \
025         "popfl"                             \
026         : "=m" (prev->thread.esp), "=m" (prev->thread.eip), \
027         "=a" (last), "=S" (esi), "=D" (edi)   \
028         : "m" (next->thread.esp), "m" (next->thread.eip), \
029         "2" (prev), "d" (next));              \
030 } while (0)
-----
```

第 12 行: `FASTCALL` 宏解析 `__attribute__((fastcall))`, 它强制将参数传入寄存器, 而不是栈。

第 15~16 行: `do {} while (0)` 结构使得这个宏拥有两个局部变量 `esi` 和 `edi`。注意: `esi` 和 `edi` 只是用熟悉的名字命名的局部变量。

current 和 task 结构

分析内核时, 无论何时需要获取或者存储正在给定处理器上运行的任务 (或进程) 的信息, 都必须利用全局变量 `current` 来引用其进程结构。例如, `current->pid` 包含进程的 ID。Linux

可以用一个既快速又巧妙的方法来引用当前进程的进程结构。

进程创建时，每个进程都被分配 8 K 的连续内存。(Linux 2.6 有一个编译时选项，通过这个选项可以使用 4 K 内存，而不是 8 K) 这个 8 K 的段由进程结构和为给定进程所设置的内核栈占用。当进程创建时，Linux 把进程结构放置在 8 K 内存的低端，而内核栈的指针则从高端开始。当数据被压到栈中时，内核栈的指针（具体是对 x86 和 PPC 的 `r1` 来说）随着数据的压入而减小。由于 8 K 的内存区是页对齐的，所以，该内存区的起始地址（用十六进制标识）总是以 0x000 结束（4 k 字节的倍数）。

读者可能已经猜到了，Linux 用来引用当前 task 结构的巧妙方法是用 0xffff_f000 和栈指针的内容进行“与”操作。通过让通用寄存器 2 保存当前进程的指针，PPC Linux 内核的最新版本把这个操作向前推进了一步。

第 17 行和第 30 行：`asm volatile()`^①这种形式封装了一个内联汇编代码块，`volatile` 关键字确保编译器不会以任何方式改变（优化）例程。

第 17~18 行：把寄存器 `flags` 和 `ebp` 压入栈中。（注意，此时仍然在使用和进程 `prev` 相关的栈）。

第 19 行：把当前进程的栈指针 `esp` 保存到 `prev` 的 task 结构。

第 20 行：把栈指针从 `next` 的 task 结构移动至当前处理器的 `esp`。

注意 迄今为止，还只是在定义上进行了上下文切换。

现在，我们有了一个新的内核栈，因而，任何对 `current` 的引用均被指向新的（即 `next`）的 task 结构。

第 21 行：将 `prev` 进程的返回地址保存到其 task 结构。当 `prev` 进程重新启动时，就从该地址恢复执行。

第 22 行：把返回地址（从 `__switch_to()` 返回的地址）压入栈中，这是来自于 `next` 的 `eip`。当进程被停止或者最近一次被抢占时，`eip` 将被保存到该进程的 task 结构中（在第 21 行）。

第 23 行：跳转到 C 函数 `__switch_to()`，更新下列信息。

- ☐ 下一个含有内核栈指针的线程结构。
- ☐ 当前处理器的线程局部存储器描述符。
- ☐ `prev` 和 `next` 的 `fs` 及 `gs`（如果需要的话）。
- ☐ 调试寄存器（如果需要的话）。
- ☐ I/O 位图（如果需要的话）。

然后，`__switch_to()` 返回更新之后的 `prev` 的进程结构。

第 24~25 行：从新的即 `next` 进程的内核栈弹出基址指针和标志寄存器。

第 26~29 行：这是传给内联汇编例程的输入参数和输出参数。关于对这些参数所施加的限制，

① 有关 `volatile` 的更多信息请参阅第 2 章。

详情请参阅 2.4 节。

第 29 行：依靠神奇的汇编代码，prev 被返回到 eax 中，它在顺序上是第三个参数。换句话说，输入参数 prev 作为输出参数 last 传给 switch_to() 宏。

因为 switch_to() 是一个宏，它内嵌在 context_switch() 中调用它的代码里执行。它不会像通常的函数那样返回。

为了清楚起见，注意，switch_to() 把 prev 传回 eax 寄存器，然后，在 context_switch() 中继续执行，其下一条指令是返回 prev (kernel/sched.c 的第 1074 行)。这使得 context_switch() 传回一个指向上一次运行的进程的指针。

2. 跟踪 PPC 的 context_switch()

要使 context_switch() 的 PPC 代码和 x86 有相同的效果，就必须稍微多做一些工作。和 x86 体系结构中的 cr3 寄存器不同，PPC 使用散列函数指向上下文环境。为 switch_mm() 编写的下列代码涉及了这些散列函数，第 4 章中详细解释了这些函数。

此处是 switch_mm() 例程，它依次调用 set_context() 例程。

```
-----
#include/asm-ppc/mmu_context.h
155 static inline void switch_mm(struct mm_struct *prev, struct
mm_struct *next, struct task_struct *tsk)
156 {
157     tsk->thread.pgdir = next->pgd;
158     get_mmu_context(next);
159     set_context(next->context, next->pgd);
160 }
-----
```

第 157 行：新线程的页全局目录（段寄存器）指向 next->pgd 指针。

第 158 行：传入 switch_mm() 的 mm_struct 的 context 字段 (next->context) 被更新为适当的上下文值。上下文的信息来自于对变量 context_map[] 的全局引用，context_map[] 含有一组位图字段。

第 159 行：此处是对汇编例程 set_context 的调用。下面是这个例程的代码以及对它的讨论，当第 1468 行的 blr 指令执行时，代码将返回到 switch_mm 例程。

```
-----
/arch/ppc/kernel/head.S
1437 _GLOBAL(set_context)
1438 mulli r3,r3,897 /* multiply context by skew factor */
1439 rlwinm r3,r3,4,8,27 /* VSID = (context & 0xffff) << 4 */
1440 addis r3,r3,0x6000 /* Set Ks, Ku bits */
1441 li r0,NUM_USER_SEGMENTS
1442 mtctr r0
...
1457 3: isync
...
1461 mtsrin r3,r4
1462 addi r3,r3,0x111 /* next VSID */
1463 rlwinm r3,r3,0,8,3 /* clear out any overflow from VSID field */
-----
```

```

1464 addis r4,r4,0x1000 /* address of next segment */
1465 bdnz 3b
1466 sync
1467 isync
1468 blr
-----

```

第 1437~1440 行：经由 r3 传递给 set_context() 的 mm_struct 的 context 字段 (next->context) 为 PPC 的分段安装散列函数。

第 1461~1465 行：经由 r4 传递给 set_context() 的 mm_struct 的 pgd 字段 (next->pgd) 指向段寄存器。

分段是 PPC 内存管理的基础 (参阅第 4 章)。当从 set_context() 返回时，将把 mm_struct 类型的 next 初始化成合适的内存区域并返回到 switch_mm()。

3. 跟踪 switch_to() 的 PPC 踪迹

PPC 对 switch_to() 的实现必然和 x86 的 switch_to() 调用完全相同；它接受 current 和 next 的进程指针，并返回指向前一个运行进程的指针。

```

-----
include/asm-ppc/system.h
88 extern struct task_struct *__switch_to(struct task_struct *,
89   struct task_struct *);
90 #define switch_to(prev, next, last)
91 ((last) = __switch_to((prev), (next)))
92
91
92 struct thread_struct;
93 extern struct task_struct *_switch(struct thread_struct *prev,
94   struct thread_struct *next);
-----

```

在第 88 行，__switch_to() 将其参数视为 task_struct 类型的，而在第 93 行，_switch() 将其参数视为 thread_struct 类型的。这是因为，task_struct 中的线程入口含有特定线程所关心的、与体系结构相关的处理器寄存器信息。现在，先来分析 __switch_to() 的实现。

```

-----
/arch/ppc/kernel/process.c
200 struct task_struct *__switch_to(struct task_struct *prev,
   struct task_struct *new)
201 {
202   struct thread_struct *new_thread, *old_thread;
203   unsigned long s;
204   struct task_struct *last;
205   local_irq_save(s);
206   ...
247   new_thread = &new->thread;
248   old_thread = &current->thread;
249   last = _switch(old_thread, new_thread);
250   local_irq_restore(s);
251   return last;
252 }
-----

```


第 205 行：上下文切换前禁止中断。

第 247~248 行：仍然运行在老线程的上下文中，把指向线程结构的指针传给 `_switch()` 函数。

第 249 行：`_switch()` 是为完成两个线程结构的切换而调用的汇编程序（参见下一节）。

第 250 行：上下文切换后开启中断。

为了更好地理解 PPC 线程内部的哪些信息必须被交换，必须考查在第 249 行传入的 `thread_struct`。

回想一下对 x86 上下文切换所做的研究，直到指向一个新的内核栈时，上下文切换才会正式生效。在 PPC 中，让上下文切换正式生效的这关键一步正是发生在 `_switch()` 中。

● 追踪 PPC 的 `_switch()` 代码

按照惯例，PPC 中 C 函数的参数（从左至右）分别保存在 `r3`、`r4`、`r5...r12` 中。进入 `switch()` 时，`r3` 指向当前进程的 `thread_struct`，`r4` 指向新进程的 `thread_struct`。

```
-----
/arch/ppc/kernel/entry.S
437 _GLOBAL(_switch)
438 stwu r1,-INT_FRAME_SIZE(r1)
439 mflr r0
440 stw r0,INT_FRAME_SIZE+4(r1)
441 /* r3-r12 are caller saved -- Cort */
442 SAVE_NVGPRS(r1)
443 stw r0,_NIP(r1) /* Return to switch caller */
444 mfmsr r11
...
458 1: stw r11,_MSR(r1)
459 mfcrr r10
460 stw r10,_CCR(r1)
461 stw r1,KSP(r3) /* Set old stack pointer */
462
463 tophys(r0,r4)
464 CLR_TOP32(r0)
465 mtspr SPRG3,r0/* Update current THREAD phys addr */
466 lwz r1,KSP(r4) /* Load new stack pointer */
467 /* save the old current 'last' for return value */
468 mr r3,r2
469 addi r2,r4,-THREAD /* Update current */
...
478 lwz r0,_CCR(r1)
479 mtcrrf 0xFF,r0
480 REST_NVGPRS(r1)
481
482 lwz r4,_NIP(r1) /* Return to _switch caller in new task */
483 mtlr r4
484 addi r1,r1,INT_FRAME_SIZE
485 blr
-----
```

逐字节的为新线程换出前一个 `thread_struct` 的机制作为练习留给大家。然而，值得注意的是，`_switch()` 中对 `r1`、`r2`、`r3`、`SPRG3` 和 `r4` 的使用参见这种操作的基本知识。

第 438~460 行：上下文环境被保存到和当前栈指针 r1 相关的当前栈中。

第 461 行：整个环境被存入经由 r3 传入的当前 thread_struct 的指针。

第 463~465 行：SPRG3 被更新为指向新进程的线程结构。

第 466 行：KSP 是 task 结构 (r4) 的偏移，新进程的内核栈指针指向这个 task 结构。现在，栈指针 r1 可以用这个值 KSP 来更新。(这一步是 PPC 上下文切换的要点。)

第 468 行：指向前一个进程的当前指针从 _switch() 返回后放在 r3 中。这表示上一个进程。

第 469 行：用指向新进程结构的指针 (r4) 来更新当前指针 (r2)。

第 478~486 行：从新栈中恢复环境的剩余部分，把 r3 中的前一个进程的结构返回给调用程序。

对 context_switch() 的分析到这里就结束了。下面的代码表示：随着被 schedule() 中的 context_switch 调用，处理器交换 prev 和 next 这两个进程。

```
-----
kernel/sched.c
1709  prev = context_switch(rq, prev, next);
-----
```

现在，prev 指向了刚才切换掉的进程，next 则指向了当前进程。

既然已经讨论了在 Linux 内核中如何调度进程，接下来就考查如何告知进程它将要被调度，即什么因素导致了 schedule() 的调用，什么因素又导致了一个进程不得不为另一个进程让出 CPU？

7.1.3 让出 CPU

只需调用 schedule() 就可使进程自动放弃 CPU。在想要睡眠或等待信号发生的内核代码以及设备驱动程序中，这个函数用得非常普遍^①。若其他进程想要连续地使用 CPU，系统定时器就必须告诉它们让出 CPU。Linux 内核定期地夺取 CPU，以此来强制停止活跃的进程，然后执行许多基于定时器的任务。scheduler_tick() 就是其中之一，该内核函数强迫一个进程放弃 CPU。如果一个进程运行时间过长，内核就不会把 CPU 的控制权交还给这个进程，而是改为选择另外一个进程。现在，给读者介绍 scheduler_tick() 如何判断当前进程是否必须放弃 CPU：

```
-----
kernel/sched.c
1981 void scheduler_tick(int user_ticks, int sys_ticks)
1982 {
1983     int cpu = smp_processor_id();
1984     struct cpu_usage_stat *cpustat = &kstat_this_cpu.cpustat;
1985     runqueue_t *rq = this_rq();
1986     task_t *p = current;
1987
1988     rq->timestamp_last_tick = sched_clock();
1989
-----
```

^① Linux 规范指出：在持有自旋锁的时候不能调用调度程序，因为这可能引起系统死锁。这是忠告！

```

1990  if (rcu_pending(cpu))
1991      rcu_check_callbacks(cpu, user_ticks);
-----

```

第 1981~1986 行：这个代码块初始化 `scheduler_tick()` 函数所必需的数据结构。`cpu`、`cpu_usage_stat` 和 `rq` 分别被设置为处理器 ID、CPU 状态和当前处理器的运行队列。`p` 是一个指针，指向正在使用 CPU 的当前进程。

第 1988 行：运行队列的最后一个节拍被设置成以纳秒为单位的当前时间。

第 1990~1991 行：在 SMP 系统中，必须检查是否有未完成的 RCU 需要执行，如果有的话就通过 `rcu_check_callbacks()` 来执行。

```

-----
kernel/sched.c
1993  /* note: this timer irq context must be accounted for as well */
1994  if (hardirq_count() - HARDIRQ_OFFSET) {
1995      cpustat->irq += sys_ticks;
1996      sys_ticks = 0;
1997  } else if (softirq_count()) {
1998      cpustat->softirq += sys_ticks;
1999      sys_ticks = 0;
2000  }
2001
2002  if (p == rq->idle) {
2003      if (atomic_read(&rq->nr_iowait) > 0)
2004          cpustat->iowait += sys_ticks;
2005      else
2006          cpustat->idle += sys_ticks;
2007      if (wake_priority_sleeper(rq))
2008          goto out;
2009      rebalance_tick(cpu, rq, IDLE);
2010      return;
2011  }
2012  if (TASK_NICE(p) > 0)
2013      cpustat->nice += user_ticks;
2014  else
2015      cpustat->user += user_ticks;
2016  cpustat->system += sys_ticks;
-----

```

第 1994~2000 行：`cpustat` 跟踪内核的统计信息，并根据已经发生的系统节拍数来更新硬件中断和软件中断的统计信息。

第 2002~2011 行：如果当前没有进程运行，则认真地检查是否有进程正在等待 I/O。如果有就增加 CPU 的 I/O 等待统计计数；否则，增加 CPU 的空闲统计计数。在单处理器系统中，`rebalance_tick()` 毫无意义，但在多处理器系统中，由于这个 CPU 没有事情可做，`rebalance_tick()` 就会尽力对当前的 CPU 进行负载平衡。

第 2012~2016 行：这个代码块完成更多 CPU 统计信息的收集。如果当前进程被礼让，就增加 CPU 的 `nice` 计数器；否则，增加用户的节拍计数器。最后，增加 CPU 的系统节拍计数器。

```

-----
kernel/sched.c
2019  if (p->array != rq->active) {
2020      set_tsk_need_resched(p);
2021      goto out;
2022  }
2023  spin_lock(&rq->lock);
-----

```

第 2019~2022 行：此处指明为什么要把指向优先级数组的指针存入进程的 `task_struct`。调度程序检查当前进程是否不再活跃。如果进程已经到期，调度程序将设置进程的重新调度标志，并跳转到 `scheduler_tick()` 函数的末尾。在那里（第 2092~2093 行），由于还没有活跃的进程，调度程序就尝试对这个 CPU 进行负载平衡。在当前进程能够调度自己或从一个成功的运行返回之前，调度程序在夺取 CPU 控制权时就会发生这种情况。

第 2023 行：得知当前进程正在运行，它没有到期并一直存在。现在，调度程序想给另外的进程让出 CPU 的控制权；它必须做的第一件事情就是获得运行队列锁。

```

-----
kernel/sched.c
2024  /*
2025  * The task was running during this tick - update the
2026  * time slice counter. Note: we do not update a thread's
2027  * priority until it either goes to sleep or uses up its
2028  * timeslice. This makes it possible for interactive tasks
2029  * to use up their timeslices at their highest priority levels.
2030  */
2031  if (unlikely(rt_task(p))) {
2032      /*
2033       * RR tasks need a special form of timeslice management.
2034       * FIFO tasks have no timeslices.
2035       */
2036      if ((p->policy == SCHED_RR) && !--p->time_slice) {
2037          p->time_slice = task_timeslice(p);
2038          p->first_time_slice = 0;
2039          set_tsk_need_resched(p);
2040
2041          /* put it at the end of the queue: */
2042          dequeue_task(p, rq->active);
2043          enqueue_task(p, rq->active);
2044      }
2045      goto out_unlock;
2046  }
-----

```

第 2031~2046 行：对于调度程序来说，最简单的情形莫过于当前进程是实时进程。实时进程总是拥有比其他任何进程都高的优先级。如果该进程是 FIFO 进程并且正在运行，就应该继续它的操作，因此，直接跳转到函数的末尾并释放运行队列锁；如果当前进程是时间片轮转（round-robin）的实时进程，就减小其时间片；如果进程没有更多的时间片，就应该调度另一个

轮转实时进程。通过 `task_timeslice()` 为当前进程计算新的时间片。接着，重新设置该进程的第一个时间片。然后，进程被标记为需要重新调度，最后，从运行队列的活跃数组删除该进程并把它添加回活跃数组，该进程被放到轮转实时进程链表的末尾。此时，调度程序跳转到函数的末尾并释放运行队列锁。

```
-----
kernel/sched.c
2047  if (!--p->time_slice) {
2048      dequeue_task(p, rq->active);
2049      set_tsk_need_resched(p);
2050      p->prio = effective_prio(p);
2051      p->time_slice = task_timeslice(p);
2052      p->first_time_slice = 0;
2053
2054      if (!rq->expired_timestamp)
2055          rq->expired_timestamp = jiffies;
2056      if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
2057          enqueue_task(p, rq->expired);
2058          if (p->static_prio < rq->best_expired_prio)
2059              rq->best_expired_prio = p->static_prio;
2060      } else
2061          enqueue_task(p, rq->active);
2062  } else {
-----
```

第 2047~2061 行：此处，调度程序知道当前进程不是实时进程。它减小进程的时间片，在这里，进程的时间片终会被耗尽到 0。调度程序从活跃数组中删除进程并设置它的需要重新调度标志。然后重新计算进程的优先级，并重新设置它的时间片。优先级的计算和时间片的重置都需要考虑进程之前的活动^①。如果运行队列的到期时间戳为 0（通常发生在运行队列的活跃数组上没有更多进程的时候），就把运行队列的到期时间戳设置为 `jiffies`。

jiffies

`jiffies` 是 32 位的变量，计算系统自引导以来发生的节拍数。在一个 100Hz 的系统中，这个数回绕到 0 大约得花 497 天。第 20 行的宏是以 `u64` 类型访问这个值的建议方法。在 `include/jiffies.h` 中也有一些用于检测的宏。

```
-----
include/linux/jiffies.h
017  extern unsigned long volatile jiffies;
020  u64 get_jiffies_64(void);
-----
```

通常，通过把交互式进程重新放回运行队列的活跃优先级数组来给予它们一些照顾，这便是第 2060 行的 `else` 子句。然而，也不应该为此饿死到期进程。我们使用 `EXPIRED_STARVING()`

^① 参见 `effective_prio()` 和 `task_timeslice()`。

(参见第 1968 行的 EXPIRED_STARVING) 来判断到期进程等待 CPU 的时间是否过长。如果第一个到期进程正在等待不合理的时间量, 或者如果到期数组含有优先级高于当前进程的进程, 函数 EXPIRED_STARVING() 就返回真。等待的时间数合理与否与负载有关, 运行的进程数目越多, 活跃数组和到期数组的交换次数就越少。

如果进程不是交互式的, 或者到期进程正处于饥饿状态, 调度程序将获取当前进程并把它放入运行队列的到期优先级数组中。如果当前进程的静态优先级高于到期运行队列中优先级最高的进程的优先级, 就更新运行队列以反映到期数组现在的优先级比之前的优先级更高。(注意, 在 Linux 中, 进程的优先级越高, 编号越小, 因此, 代码中用了 <。)

```
-----
kernel/sched.c
2062  } else {
2063      /*
2064       * Prevent a too long timeslice allowing a task to monopolize
2065       * the CPU. We do this by splitting up the timeslice into
2066       * smaller pieces.
2067       *
2068       * Note: this does not mean the task's timeslices expire or
2069       * get lost in any way, they just might be preempted by
2070       * another task of equal priority. (one with higher
2071       * priority would have preempted this task already.) We
2072       * requeue this task to the end of the list on this priority
2073       * level, which is in essence a round-robin of tasks with
2074       * equal priority.
2075       *
2076       * This only applies to tasks in the interactive
2077       * delta range with at least TIMESLICE_GRANULARITY to requeue.
2078       */
2079      if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
2080          p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
2081          (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
2082          (p->array == rq->active)) {
2083
2084          dequeue_task(p, rq->active);
2085          set_tsk_need_resched(p);
2086          p->prio = effective_prio(p);
2087          enqueue_task(p, rq->active);
2088      }
2089  }
2090 out_unlock:
2091  spin_unlock(&rq->lock);
2092 out:
2093  rebalance_tick(cpu, rq, NOT_IDLE);
2094 }
-----
```

第 2079~2089 行: 调度程序执行之前的最后一种情形是当前进程正在运行并且还有剩余的时间片可运行。调度程序必须确保拥有大块时间片的进程不会独占 CPU。如果进程是交互式的, 拥

有大于 `TIMESLICE_GRANULARITY` 的时间片且是活跃的，那么调度程序将从活跃队列删除它。于是，进程让它的重新调度标志置位，在重新计算其优先级后被放回运行队列的活跃数组。这样做的目的是确保存在拥有大块时间片的具有某一优先级的进程时，不会饿死同一优先级的其他进程。

第 2092~2094 行：调度程序完成运行队列的重新排列并对它解锁；如果是在 SMP 系统中执行，那么还要进行负载平衡。

如何通过 `scheduler_tick()` 把进程标记为重新调度，以及如何通过 `schedule()` 来调度进程，这两者结合起来说明了在 Linux 2.6 内核中调度程序是如何运转的。接下来将深入研究“优先级”对调度程序的意义。

1. 动态优先级的计算

前一节没有介绍如何计算进程动态优先级的细节。进程的优先级基于它先前的行为以及用户指定的 `nice` 值。函数 `recalc_task_prio()` 用来为进程确定新的动态优先级：

```
-----
kernel/sched.c
381 static void recalc_task_prio(task_t *p, unsigned long long now)
382 {
383     unsigned long long __sleep_time = now - p->timestamp;
384     unsigned long sleep_time;
385
386     if (__sleep_time > NS_MAX_SLEEP_AVG)
387         sleep_time = NS_MAX_SLEEP_AVG;
388     else
389         sleep_time = (unsigned long)__sleep_time;
390
391     if (likely(sleep_time > 0)) {
392         /*
393          * User tasks that sleep a long time are categorised as
394          * idle and will get just interactive status to stay active &
395          * prevent them suddenly becoming cpu hogs and starving
396          * other processes.
397          */
398         if (p->mm && p->activated != -1 &&
399             sleep_time > INTERACTIVE_SLEEP(p)) {
400             p->sleep_avg = JIFFIES_TO_NS(MAX_SLEEP_AVG -
401                 AVG_TIMESLICE);
402             if (!HIGH_CREDIT(p))
403                 p->interactive_credit++;
404         } else {
405             /*
406              * The lower the sleep avg a task has the more
407              * rapidly it will rise with sleep time.
408              */
409             sleep_time *= (MAX_BONUS - CURRENT_BONUS(p)) ? : 1;
410
411             /*
412              * Tasks with low interactive_credit are limited to
413              * one timeslice worth of sleep avg bonus.
```

```

414     */
415     if (LOW_CREDIT(p) &&
416         sleep_time > JIFFIES_TO_NS(task_timeslice(p)))
417         sleep_time = JIFFIES_TO_NS(task_timeslice(p));
418
419     /*
420     * Non high_credit tasks waking from uninterruptible
421     * sleep are limited in their sleep_avg rise as they
422     * are likely to be cpu hogs waiting on I/O
423     */
424     if (p->activated == -1 && !HIGH_CREDIT(p) && p->mm) {
425         if (p->sleep_avg >= INTERACTIVE_SLEEP(p))
426             sleep_time = 0;
427         else if (p->sleep_avg + sleep_time >=
428                 INTERACTIVE_SLEEP(p)) {
429             p->sleep_avg = INTERACTIVE_SLEEP(p);
430             sleep_time = 0;
431         }
432     }
433
434     /*
435     * This code gives a bonus to interactive tasks.
436     *
437     * The boost works by updating the 'average sleep time'
438     * value here, based on ->timestamp. The more time a
439     * task spends sleeping, the higher the average gets -
440     * and the higher the priority boost gets as well.
441     */
442     p->sleep_avg += sleep_time;
443
444     if (p->sleep_avg > NS_MAX_SLEEP_AVG) {
445         p->sleep_avg = NS_MAX_SLEEP_AVG;
446         if (!HIGH_CREDIT(p))
447             p->interactive_credit++;
448     }
449 }
450 }
451
452 p->prio = effective_prio(p);
453 }

```

第 386~389 行：基于时间 `now` 来计算进程 `p` 已经睡眠了多长时间，并把计算得到的值赋给 `sleep_time`，`sleep_time` 的最大值是 `NS_MAX_SLEEP_AVG` (`NS_MAX_SLEEP_AVG` 默认为 10 毫秒)。

第 391~404 行：如果进程 `p` 已经睡眠了，那么首先检查它是否睡眠了足够长的时间（这个时间应足以把进程归类为交互式进程）。如果睡眠时间足够长，当 `sleep_time > INTERACTIVE_SLEEP(p)` 时，就把进程的平均睡眠时间调整为一个固定值，如果 `p` 还不能归类为交互式进程，就增加 `p` 的 `interactive_credit`。

第 405~410 行：给予平均睡眠时间短的进程更多的睡眠时间。

第 411~418 行：如果进程是 CPU 消耗型（CPU 密集型）的，它就被归类为非交互式的，那么就限定进程的时间片至多相当于平均睡眠时间的奖励值。

第 419~432 行：还没有归类为交互式的（非 HIGH_CREDIT 的），从不可中断的睡眠状态醒来的进程，将其平均睡眠时间限定为 INTERACTIVE()。

第 434~450 行：将重新计算过的 sleep_time 加到进程的平均睡眠时间上，要确保它不会超过 NS_MAX_SLEEP_AVG。如果进程被认为不是交互式的，但却睡眠了最大的时间甚至更长的时间，就增加其交互信用。

第 452 行：最后，根据刚刚计算出来的进程 p 的 sleep_avg 字段，利用 effective_prio() 来设置 p 的优先级，该函数通过把 0..MAX_SLEEP_AVG 的平均睡眠时间换算成 -5~+5 的数来完成优先级的设置。因此，依赖于其先前的行为，一个静态优先级为 70 的进程能够拥有一个 65~85 的动态优先级。

非实时进程的进程优先级在 101~140 的范围内变化。即使以非常高的优先级运行的进程（105 甚至更小一点），也不能超越实时进程的界限。显然，高优先级、高交互信用的进程决不可能拥有一个低于 101 的动态优先级（在默认配置中实时进程的优先级涵盖范围为 0~100）。

2. 从运行队列删除进程

前面已经讨论了如何通过分叉把进程插入调度程序，以及如何将进程从 CPU 运行队列中的活跃优先级数组移动至到期优先级数组。然而，究竟如何从运行队列删除进程呢？

主要有两种方法可以从运行队列中删除进程。

❑ 进程被内核抢占且它并不处于运行状态，并且进程没有挂起的信号（参见 kernel/sched.c 中的第 2240 行）。

❑ 在 SMP 机器中，进程能够从一个运行队列移出放到另一个运行队列中（参见 kernel/sched.c 中的第 3384 行）。

第一种情形通常发生在进程把自己放到等待队列上睡眠之后，schedule() 被调用之时。进程把自己标记为非运行状态（TASK_INTERRUPTIBLE、TASK_UNINTERRUPTIBLE、TASK_STOPPED 等），并且从运行队列彻底删除它以后，内核不再考虑该进程对 CPU 的访问。

进程被移动到另一个运行队列的情形在 Linux 内核的 SMP 部分处理，此处不作研究。

现在，我们来追踪如何通过 deactivate_task() 从运行队列删除进程：

```
-----
kernel/sched.c
507 static void deactivate_task(struct task_struct *p, runqueue_t *rq)
508 {
509     rq->nr_running--;
510     if (p->state == TASK_UNINTERRUPTIBLE)
511         rq->nr_uninterruptible++;
512     dequeue_task(p, p->array);
513     p->array = NULL;
514 }
-----
```

第 509 行：由于 p 不再运行，调度程序首先减小运行队列的运行进程数。

第 510~511 行：如果进程是不可中断的，就增加运行队列上不可中断进程的计数。相应地，当不可中断的进程醒来时则减小该计数。（参见 kernel/sched.c 的 `try_to_wake_up()` 函数中的第 824 行）。

第 512~513 行：现在，运行队列的统计信息全部更新了，接下来，将正式从运行队列删除进程 `p`。内核利用 `p->array` 字段检测进程是否正在运行以及是否在运行队列中。由于 `p->array` 不再属于上述这两种情况，因此把它设置为 `NULL`。

还有一些运行队列的管理工作需要完成；让我们来考查一下 `dequeue_task()` 的细节：

```
-----
kernel/sched.c
303 static void dequeue_task(struct task_struct *p, prio_array_t *array)
304 {
305     array->nr_active--;
306     list_del(&p->run_list);
307     if (list_empty(array->queue + p->prio))
308         __clear_bit(p->prio, array->bitmap);
309 }
-----
```

第 305 行：调整优先级数组中的活跃进程数，进程 `p` 不在到期数组上，就在活跃数组上。

第 306~308 行：根据 `p` 的优先级，从优先级数组中的相应进程链表中删除进程 `p`。如果删除操作导致链表为空，就必须清除优先级数组位图中的相应位，以表示再也没有任何进程的优先级为 `p->prio()`。

由于 `p->run_list` 是一个 `list_head` 结构，并有两个分别指向链表中前一个元素和下一个元素的指针，因此，仅用 `list_del()` 便可完成所有的删除操作。

至此，进程 `p` 从运行队列中消失了，完全不再是活跃进程，如果该进程处于 `TASK_NTERRUPTIBLE` 或 `TASK_UNINTERRUPTIBLE` 状态的，它还有被唤醒并回到运行队列的可能。如果进程处于 `TASK_STOPPED`、`TASK_ZOMBIE` 或 `TASK_DEAD` 状态的话，它的所有结构都要被删除和丢弃得干干净净。

7.2 内核抢占

内核抢占指从一个进程切换到另一个进程。前面已经讨论过 `schedule()` 和 `scheduler_tick()` 如何决定接下来要切换到哪一个进程，但仍然没有描述 Linux 内核如何确定进行切换的具体时机。Linux 2.6 内核引入了内核抢占机制，这意味着在任何时刻，不论是用户空间的程序还是内核空间的程序都能够被切换下来。因为内核抢占是 Linux 2.6 中的标准，下面将介绍 Linux 中内核抢占和用户抢占是如何进行的。

7.2.1 显式内核抢占

最容易理解的抢占是显式内核抢占。显式内核抢占发生在内核代码调用 `schedule()` 时的内核空间中。内核代码可通过两种方式调用 `schedule()`：直接调用 `schedule()`，或者自我阻塞。

当显式地抢占内核时，例如，设备驱动程序在等待队列 `wait_queue` 中等待时，控制权被简

单地交给调度程序，从而新的进程被选中执行。

7.2.2 隐式用户抢占

当内核处理完内核空间的进程，准备把控制权交给用户空间的进程时，它将首先查看应该把控制权交给用户空间的哪一个进程，这个进程也许不是之前将控制权交给内核的那个进程。例如，如果进程 A 调用了系统调用，那么该系统调用完成之后，内核可能把系统的控制权交给进程 B。

系统中的每个进程都有一个“需要重新调度”的标志，该标志在进程应该被重新调度的时候被设置。

```
-----
include/linux/sched.h
988 static inline void set_tsk_need_resched(struct task_struct *tsk)
989 {
990     set_tsk_thread_flag(tsk, TIF_NEED_RESCHED);
991 }
992
993 static inline void clear_tsk_need_resched(struct task_struct *tsk)
994 {
995     clear_tsk_thread_flag(tsk, TIF_NEED_RESCHED);
996 }
...
1003 static inline int need_resched(void)
1004 {
1005     return unlikely(test_thread_flag(TIF_NEED_RESCHED));
1006 }
-----
```

第 988~996 行：set_tsk_need_resched 和 clear_tsk_need_resched 是两个接口，用于设置和体系结构相关的标志 TIF_NEED_RESCHED。

第 1003~1006 行：need_resched 用于测试当前线程的标志 TIF_NEED_RESCHED 是否被设置。

如同 schedule() 和 scheduler_tick() 中描述的那样，当内核将要返回用户空间时，它将选择一个接受控制权的进程。尽管 scheduler_tick() 能够把进程标记为需要重新调度，但是这个标志只对 schedule() 有用。schedule() 反复选择一个新进程来执行，直到新选择的进程不需要重新调度为止。schedule() 完成后，新进程拥有处理器的控制权。

综上所述，当进程正在运行时，系统定时器引发一个触发 scheduler_tick() 的中断。scheduler_tick() 能够将那个进程标记为需要重新调度并把它移动到到期数组中。当内核操作完成时，scheduler_tick() 后面可能紧跟着其他中断，这样，内核将继续拥有对处理器的控制权，调用 schedule() 选择下一个要运行的进程。这样看来，scheduler_tick() 标记进程并重排运行队列，而 schedule() 选择下一个进程并传递 CPU 的控制权。

7.2.3 隐式内核抢占

Linux 2.6 中新增了对隐式内核抢占的实现。当内核进程拥有对 CPU 的控制权时，仅当其目前没有持有任何锁时，才能被另一个内核进程所抢占。每个进程有一个 preempt_count 字段，

用于标记进程是否可以被抢占。每当进程获得一次锁时，该计数增加；每当释放一次锁时，该计数减少。schedule()函数在决定接下来要运行哪个进程期间是禁止抢占的。

隐式内核抢占有两种可能性：内核代码出自禁止抢占的代码块，或者处理过程正在从中断返回内核代码。如果控制权正在从一个中断返回到内核空间，该中断调用 schedule()并以刚才描述的方式选中一个新进程。

如果内核代码出自禁止抢占的代码块，启用抢占的操作可能引起当前进程被抢占：

```
-----
include/linux/preempt.h
46 #define preempt_enable() \
47 do { \
48     preempt_enable_no_resched(); \
49     preempt_check_resched(); \
50 } while (0)
-----
```

第 46~50 行：preempt_enable()调用 preempt_enable_no_resched()，后者将当前进程的 preempt_count 减 1，然后调用 preempt_check_resched()：

```
-----
include/linux/preempt.h
40 #define preempt_check_resched() \
41 do { \
42     if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \
43         preempt_schedule(); \
44 } while (0)
-----
```

第 40~44 行：preempt_check_resched()判断当前进程是否被标记为重新调度：如果是，则调用 preempt_schedule()。

```
-----
kernel/sched.c
2328 asmlinkage void __sched preempt_schedule(void)
2329 {
2330     struct thread_info *ti = current_thread_info();
2331
2332     /*
2333      * If there is a non-zero preempt_count or interrupts are disabled,
2334      * we do not want to preempt the current task. Just return..
2335      */
2336     if (unlikely(ti->preempt_count || irqs_disabled()))
2337         return;
2338
2339 need_resched:
2340     ti->preempt_count = PREEMPT_ACTIVE;
2341     schedule();
2342     ti->preempt_count = 0;
2343
2344     /* we could miss a preemption opportunity between schedule and now */
2345     barrier();
-----
```



```

2346  if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
2347      goto need_resched;
2348  }

```

第 2336~2337 行：如果当前进程的 `preempt_count` 值仍然是正的（可能由于嵌套的 `preempt_disable()` 命令）或者当前进程禁用了中断，就把处理器的控制权交还给当前进程。

第 2340~2347 行：`preempt_count` 为 0 说明当前进程不持有任何锁并且允许 IRQ。因此，将当前进程的 `preempt_count` 设置成表明它正在经历抢占的状态，并调用 `schedule()` 来选择另一个进程。

如果进程出自需要重新调度的代码块，内核就必须确保当前进程放弃处理器是安全的。内核将检查进程的 `preempt_count` 值。如果 `preempt_count` 为 0，表示当前进程不持有锁，则 `schedule()` 被调用，一个新的进程被选中执行；如果 `preempt_count` 非 0，这时，把控制权传递给其他进程是危险的，于是，控制权被交还给当前进程，直到当前进程释放它的全部锁为止。当前进程释放锁时，要进行一个测试，判断其是否需要被重新调度。待当前进程释放它的最后一个锁后，`preempt_count` 就变成了 0，于是立刻进行调度。

7.3 自旋锁和信号量

当两个或多于两个进程请求独占地访问共享资源时，它们也许需要具备这样一种条件，即它们是在给定代码段中操作的唯一进程。Linux 内核中锁的基本形式是自旋锁。

自旋锁是因其连续地循环等待或连续地旋转等待以获得一个锁而得名。由于自旋锁的这种操作方式，禁止自旋锁中的任何代码段企图两次获得锁是非常必要的，否则将导致死锁。

对一个自旋锁进行操作前，必须初始化 `spin_lock_t`。这可通过调用 `spin_lock_init()` 来完成：

```

-----
include/linux/spinlock.h
63 #define spin_lock_init(x) \
64  do { \
65      (x)->magic = SPINLOCK_MAGIC; \
66      (x)->lock = 0; \
67      (x)->babble = 5; \
68      (x)->module = __FILE__; \
69      (x)->owner = NULL; \
70      (x)->oline = 0; \
71  } while (0)
-----

```

这段代码在第 66 行设置自旋锁为“开锁”状态（或 0）并初始化结构中的其他变量。此处，我们所关心的变量是 `(x)->lock`。

自旋锁被初始化后，可以通过调用 `spin_lock()` 或 `spin_lock_irqsave()` 来获取它。`spin_lock_irqsave()` 函数在上锁前禁止中断，而 `spin_lock()` 不会禁止中断。如果使用 `spin_lock()`，进程是有可能在上锁的代码段中被中断的。

为了在代码的临界区执行后释放锁，必须调用 `spin_unlock()` 或 `spin_unlock_irqrestore()`。`spin_unlock_irqrestore()`把中断寄存器的状态恢复成调用 `spin_lock_irq()`时的状态。

接下来考查一下 `spin_lock_irqsave()`和 `spin_unlock_irqrestore()`：

```
-----
include/linux/spinlock.h
258 #define spin_lock_irqsave(lock, flags) \
259 do { \
260     local_irq_save(flags); \
261     preempt_disable(); \
262     _raw_spin_lock_flags(lock, flags); \
263 } while (0)
...
321 #define spin_unlock_irqrestore(lock, flags) \
322 do { \
323     _raw_spin_unlock(lock); \
324     local_irq_restore(flags); \
325     preempt_enable(); \
326 } while (0)
-----
```

注意，在上锁期间是如何禁止抢占的。禁止中断可确保临界区中的任何操作不被打断。第 324 行恢复了第 260 行保存的中断请求标志。

自旋锁的缺点是频繁地循环等待锁的释放。将它们用于可快速完成的代码临界区是最好不过的。对于花费时间较多的代码区，最好使用 Linux 内核的另一个上锁工具：信号量。

信号量与自旋锁的不同之处在于，当进程企图获得一个竞争资源时，进程是睡眠的而不是忙于等待的。信号量的主要优势之一是：持有信号量的进程可以安全地阻塞，它们在 SMP 和中断中都是安全的：

```
-----
include/asm-i386/semaphore.h
44 struct semaphore {
45     atomic_t count;
46     int sleepers;
47     wait_queue_head_t wait;
48 #ifdef WAITQUEUE_DEBUG
49     long __magic;
50 #endif
51 };
-----
include/asm-ppc/semaphore.h
24 struct semaphore {
25     /*
26     * Note that any negative value of count is equivalent to 0,
27     * but additionally indicates that some process(es) might be
28     * sleeping on 'wait'.
29     */
30     atomic_t count;
```

```

31  wait_queue_head_t wait;
32  #ifdef WAITQUEUE_DEBUG
33  long __magic;
34  #endif
35  };
-----

```

这两种体系结构对信号量的实现都提供了一个指向 `wait_queue` 的指针和一个计数。计数指的是同一时刻持有信号量的进程的数目。有了信号量,就能够让多个进程同时进入代码的临界区。如果计数被初始化成 1,表示只有一个进程能够进入代码的临界区;计数为 1 的信号量称为互斥信号量 (mutex)。

信号量用 `sema_init()` 来初始化,并分别通过调用 `down()` 和 `up()` 来上锁和解锁。如果进程对一个已上锁的信号量调用 `down()`,它就会阻塞并忽略发送给它的所有信号。当然也存在函数 `down_interruptible()`,如果获得了信号量,`down_interruptible()` 就返回 0;如果进程在阻塞期间被中断,它就返回 `-EINTR`。

当进程调用 `down()` 或 `down_interruptible()` 时,信号量的 `count` 字段递减。如果 `count` 字段小于 0,那么调用 `down()` 的进程被阻塞并被添加到信号量的 `wait_queue`;如果 `count` 字段大于或等于 0,该进程将继续执行。

执行完临界区的代码之后,进程应该调用 `up()` 告知信号量自己已离开临界区。进程调用 `up()`,增加信号量的 `count` 字段,如果 `count` 大于或等于 0,则唤醒在信号量的 `wait_queue` 上等待的某个进程。

7.4 系统时钟：关于时间和定时器

为了进行调度,内核用系统时钟来获知进程运行了多长时间。第 5 章已经将系统时钟作为讨论中断的一个例子进行了介绍。此处将分析实时时钟和它的使用及其实现;首先简单概述一下时钟。

时钟是作用在处理器上的周期性信号,它使得处理器在时域中运行。处理器依靠时钟信号来获知什么时候能够执行它的下一个任务,例如将两个整数相加或从内存取数据。这种时钟信号的速率 (1.4GHz、2GHz, 等等) 曾被用于比较系统在局部电子存储方面的处理速度。

在任何给定的时刻,系统中总有几个时钟或定时器在运行。一些简单的例子包括显示在屏幕右下角的时刻 (也叫做挂钟时间)、光标耐心地在杂乱的桌面上跳动,或者笔记本电脑的屏保由于屏幕的休止而接管了屏幕。更复杂的计时例子包括音频和视频的重放、键的重复 (保持一个键的按下状态)、通信端口的运行速度有多快,以及如前面所讨论的,进程运行了多长时间。

7.4.1 实时时钟：现在几点了

挂钟时间 (wall clock time) 的 Linux 接口通过 `/dev/rtc` 设备驱动程序的 `ioctl()` 函数来实现。使用这个驱动程序的设备称为 RTC (实时时钟)。RTC^① 给计时函数提供一个 114 字节大小的

① RTC 由多个厂家生产,其中最有名的是摩托罗拉的 `mc146818` (这种 RTC 已经不再生产,而是用 Dallas `DS12885` 或同等产品来替代)。

小型用户 NVRAM。该设备的输入是一个 32.768KHz 的振荡器和一个与备用电池的连接。个别的 RTC 模型将振荡器和电池嵌入其中，而其他的 RTC 目前被嵌入处理器芯片集的外围总线控制器（例如，南桥）。RTC 不仅能报告时刻，而且也是一个能中断系统的可编程定时器，其中断频率为 2~8192Hz。RTC 也可以像闹钟那样每日都中断。下面分析 RTC 的代码：

```
-----
#include/linux rtc.h

/*
 * ioctl calls that are permitted to the /dev/rtc interface, if
 * any of the RTC drivers are enabled.
 */

70 #define RTC_AIE_ON      _IO('p', 0x01)    /* Alarm int. enable on */
71 #define RTC_AIE_OFF     _IO('p', 0x02)    /* ... off */
72 #define RTC_UIE_ON      _IO('p', 0x03)    /* Update int. enable on */
73 #define RTC_UIE_OFF     _IO('p', 0x04)    /* ... off */
74 #define RTC_PIE_ON      _IO('p', 0x05)    /* Periodic int. enable on */
75 #define RTC_PIE_OFF     _IO('p', 0x06)    /* ... off */
76 #define RTC_WIE_ON      _IO('p', 0x0f)    /* Watchdog int. enable on */
77 #define RTC_WIE_OFF     _IO('p', 0x10)    /* ... off */

78 #define RTC_ALM_SET      _IOW('p', 0x07, struct rtc_time) /* Set alarm time */
79 #define RTC_ALM_READ     _IOR('p', 0x08, struct rtc_time) /* Read alarm time */
80 #define RTC_RD_TIME      _IOR('p', 0x09, struct rtc_time) /* Read RTC time */
81 #define RTC_SET_TIME     _IOW('p', 0x0a, struct rtc_time) /* Set RTC time */
82 #define RTC_IRQP_READ    _IOR('p', 0x0b, unsigned long) /* Read IRQ rate */
83 #define RTC_IRQP_SET     _IOW('p', 0x0c, unsigned long) /* Set IRQ rate */
84 #define RTC_EPOCH_READ   _IOR('p', 0x0d, unsigned long) /* Read epoch */
85 #define RTC_EPOCH_SET    _IOW('p', 0x0e, unsigned long) /* Set epoch */
86
87 #define RTC_WKALM_SET     _IOW('p', 0x0f, struct rtc_wkalrm) /* Set wakealarm */
88 #define RTC_WKALM_RD     _IOR('p', 0x10, struct rtc_wkalrm) /* Get wakealarm */
89
90 #define RTC_PLL_GET       _IOR('p', 0x11, struct rtc_pll_info) /* Get PLL correction */
91 #define RTC_PLL_SET       _IOW('p', 0x12, struct rtc_pll_info) /* Set PLL correction */
-----
```

ioctl() 控制函数位于文件 include/linux/rtc.h 中。在本书中，对于 PPC 的体系结构来说，并不是所有对 RTC 的 ioctl() 调用都要被实现。这些控制函数都调用低层的、硬件特有的函数（如果已经实现了的话）。本节的例子中使用 RTC_RD_TIME 函数。

下面是利用 ioctl() 调用获得日期和时间的例子。这个程序只是打开驱动程序并向 RTC 硬件查询当前的日期和时间，然后将信息输出到标准错误输出终端 stderr。注意，每次只有一个用户能够访问 RTC 的驱动程序。在讨论驱动程序的代码中强调了这一点。

```
-----
Documentation/rtc.txt
/*
 * Trimmed down version of code in /Documentation/rtc.txt
 *
 */
-----
```

```

int main(void) {

int fd, retval = 0;
//unsigned long tmp, data;
struct rtc_time rtc_tm;

fd = open ("/dev/rtc", O_RDONLY);

/* Read the RTC time/date */
retval = ioctl(fd, RTC_RD_TIME, &rtc_tm);

/* print out the time from the rtc_tm variable */

close(fd);
return 0;

} /* end main */
-----

```

这段代码是/Documentation/rtc.txt 中一个完整例子的片断。在这个程序中，两行主要的代码是 open() 命令和 ioctl() 调用。open() 告诉我们要使用哪一个驱动程序 (/dev/rtc)，ioctl() 则指出一个明确的路径，该路径可到达由 RTC_RD_TIME 命令指明的 RTC 物理接口。open() 命令的驱动程序代码驻留在驱动程序源码中，在这里，讨论它的唯一意义只在于表明打开的是哪一个设备驱动程序。

7.4.2 读取 PPC 的实时时钟

内核编译时，将在内核中插入适当的代码树 (x86、PPC、MIPS，等等)。此处，对于非 x86 系统的通用 RTC 驱动程序，在源代码文件中只讨论 PPC 源码这一分支：

```

-----
/drivers/char/genrtc.c
276 static int gen_rtc_ioctl(struct inode *inode, struct file *file,
277     unsigned int cmd, unsigned long arg)
278 {
279     struct rtc_time wtime;
280     struct rtc_pll_info pll;
281
282     switch (cmd) {
283
284     case RTC_PLL_GET:
285     ...
290     case RTC_PLL_SET:
291     ...
298     case RTC_UIE_OFF: /* disable ints from RTC updates. */
299     ...
302     case RTC_UIE_ON: /* enable ints for RTC updates. */
303     ...

```

```

305 case RTC_RD_TIME: /* Read the time/date from RTC */
306
307     memset(&wtime, 0, sizeof(wtime));
308     get_rtc_time(&wtime);
309
310     return copy_to_user((void *)arg, &wtime, sizeof(wtime)) ? -EFAULT:0;
311
312 case RTC_SET_TIME: /* Set the RTC */
313     return -EINVAL;
314 }
...
353 static int gen_rtc_open(struct inode *inode, struct file *file)
354 {
355     if (gen_rtc_status & RTC_IS_OPEN)
356         return -EBUSY;
357     gen_rtc_status |= RTC_IS_OPEN;
-----

```

这段代码是针对 `ioctl` 命令集的 `case` 语句。因为以 `RTC_RD_TIME` 标志从用户空间的测试程序调用了 `ioctl`，所以控制权被传递到第 305 行。下一个调用的是第 308 行的 `get_rtc_time(&wtime)`，它位于 `rtc.h` 中（见下列代码）。在离开这个代码段前，请留意一下第 353 行。将状态设置为 `RTC_IS_OPEN`，每次只允许一个用户可以通过 `open()` 来访问驱动程序。

```

-----
include/asm-ppc/rtc.h
045 static inline unsigned int get_rtc_time(struct rtc_time *time)
046 {
047     if (ppc_md.get_rtc_time) {
048         unsigned long nowtime;
049
050         nowtime = (ppc_md.get_rtc_time)();
051
052         to_tm(nowtime, time);
053
054         time->tm_year -= 1900;
055         time->tm_mon -= 1; /* Make sure userland has a 0-based month */
056     }
057     return RTC_24H;
058 }
-----

```

内联函数 `get_rtc_time()` 通过第 50 行上的 `ppc_md.get_rtc_time` 调用结构体变量所指向的函数。早在内核初始化时，就已经在 `chrp_setup.c` 中设置了该变量。

```

-----
arch/ppc/platforms/chrp_setup.c
447 chrp_init(unsigned long r3, unsigned long r4, unsigned long r5,
448 unsigned long r6, unsigned long r7)
449 {
...
477     ppc_md.time_init = chrp_time_init;
478     ppc_md.set_rtc_time = chrp_set_rtc_time;

```



```

479  ppc_md.get_rtc_time = chrp_get_rtc_time;
480  ppc_md.calibrate_decr = chrp_calibrate_decr;
-----

```

在下列代码段中，函数 `chrp_get_rtc_time()`（第 479 行）是在 `chrp_time.c` 中被定义的。因为 CMOS 存储器中的时间信息被周期性地更新，所以，这段读代码被封装在 `for` 循环中，如果正在进行更新，则重读该代码块。

```

-----
arch/ppc/platforms/chrp_time.c
122  unsigned long __chrp chrp_get_rtc_time(void)
123  {
124      unsigned int year, mon, day, hour, min, sec;
125      int uip, i;
126      ...
141      for ( i = 0; i<1000000; i++) {
142          uip = chrp_cmos_clock_read(RTC_FREQ_SELECT);
143          sec = chrp_cmos_clock_read(RTC_SECONDS);
144          min = chrp_cmos_clock_read(RTC_MINUTES);
145          hour = chrp_cmos_clock_read(RTC_HOURS);
146          day = chrp_cmos_clock_read(RTC_DAY_OF_MONTH);
147          mon = chrp_cmos_clock_read(RTC_MONTH);
148          year = chrp_cmos_clock_read(RTC_YEAR);
149          uip |= chrp_cmos_clock_read(RTC_FREQ_SELECT);
150          if ((uip & RTC_UIP)==0) break;
151      }
152      if (!(chrp_cmos_clock_read(RTC_CONTROL)
153          & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
154      {
155          BCD_TO_BIN(sec);
156          BCD_TO_BIN(min);
157          BCD_TO_BIN(hour);
158          BCD_TO_BIN(day);
159          BCD_TO_BIN(mon);
160          BCD_TO_BIN(year);
161      }
162      ...
054  int __chrp chrp_cmos_clock_read(int addr)
055  {  if (nvram_as1 != 0)
056      outb(addr>>8, nvram_as1);
057      outb(addr, nvram_as0);
058      return (inb(nvram_data));
059  }
-----

```

最后，在 `chrp_get_rtc_time()` 中，使用函数 `chrp_cmos_clock_read`，从 RTC 设备读取时间结构各成员的值，然后用 `rtc_tm` 结构格式化并返回这些值，`rtc_tm` 结构被传回用户空间测试程序的 `ioctl` 回调。

7.4.3 读取 x86 的实时时钟

在 x86 系统中，读取 RTC 的方法类似于 PPC 中的方法，但是更简洁、更健壮一些。继续分

析打开的驱动程序/dev/rtc，但是这次，编译程序已经为 x86 体系结构编译了文件 rtc.c。此处讨论 x86 源码分支。

```
-----
drivers/char/rtc.c
...
352 static int rtc_do_ioctl(unsigned int cmd, unsigned long arg, int kernel)
353 {
...
switch (cmd) {
...
482 case RTC_RD_TIME: /* Read the time/date from RTC */
483 {
484     rtc_get_rtc_time(&wtime);
485     break;
486 }
...
1208 void rtc_get_rtc_time(struct rtc_time *rtc_tm)
1209 {
...
1238     spin_lock_irq(&rtc_lock);
1239     rtc_tm->tm_sec = CMOS_READ(RTC_SECONDS);
1240     rtc_tm->tm_min = CMOS_READ(RTC_MINUTES);
1241     rtc_tm->tm_hour = CMOS_READ(RTC_HOURS);
1242     rtc_tm->tm_mday = CMOS_READ(RTC_DAY_OF_MONTH);
1243     rtc_tm->tm_mon = CMOS_READ(RTC_MONTH);
1244     rtc_tm->tm_year = CMOS_READ(RTC_YEAR);
1245     ctrl = CMOS_READ(RTC_CONTROL);
...
1249     spin_unlock_irq(&rtc_lock);
1250
1251     if (!(ctrl & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
1252     {
1253         BCD_TO_BIN(rtc_tm->tm_sec);
1254         BCD_TO_BIN(rtc_tm->tm_min);
1255         BCD_TO_BIN(rtc_tm->tm_hour);
1256         BCD_TO_BIN(rtc_tm->tm_mday);
1257         BCD_TO_BIN(rtc_tm->tm_mon);
1258         BCD_TO_BIN(rtc_tm->tm_year);
1259     }
-----
```

测试程序在对驱动程序 rtc.c 的调用中使用了 ioctl() 的 RTC_RD_TIME 标志。接着，ioctl 的 switch 语句用 RTC 的 CMOS 存储器中的内容来填充时间结构。此处是如何读取 RTC 硬件的 x86 实现：

```
-----
include/asm-i386/mc146818rtc.h
...
018 #define CMOS_READ(addr) ({ \
019     outb_p((addr), RTC_PORT(0)); \
```

```
020    inb_p(RTC_PORT(1)); \
021  })
```

7.5 小结

本章介绍了 Linux 的调度程序、内核抢占，以及系统时钟和定时器。

更具体地说，介绍了下列内容。

- 简单介绍了 Linux 2.6 的新调度程序及其新特性。
- 描述了调度程序如何从所有可供选择的进程中选择下一个进程，以及调度程序选择进程时所使用的算法。
- 讨论了调度程序用来真实地交换进程的上下文切换，追踪了从切换函数一直到低层的体系结构特有代码的全过程。
- 介绍了 Linux 中的进程如何通过调用 `schedule()` 为其他进程让出 CPU，以及接下来内核如何把该进程标记为“将要被调度”的进程。
- 深入研究了 Linux 内核如何基于某个进程先前的行为来计算其动态优先级，以及最终如何从调度队列删除进程。
- 分别介绍了隐式和显式的用户级和内核级的抢占，以及在 Linux 2.6 内核中如何处理每一种抢占。
- 分析了定时器和系统时钟，以及在 x86 和 PPC 这两种体系结构中系统时钟分别是如何实现的。

7.6 习题

1. Linux 如何通知调度程序周期性地运行？
2. 请描述交互式进程和非交互式进程之间的区别。
3. 实时进程对于调度程序有何特殊之处？
4. 进程用完调度程序的节拍时会发生什么？
5. O(1)调度程序的优势是什么？
6. 调度程序使用何种数据结构来管理在系统中运行的进程的优先级？
7. 如果在持有自旋锁的情况下调用 `schedule()`，会发生什么？
8. 内核如何确定一个内核进程是否可以被隐式地抢占？

第 8 章

内核引导

本章内容

- BIOS 和 Open Firmware
- 引导加载程序
- 与体系结构相关的内存初始化
- 原始的 RAM 盘
- 开始: `start_kernel()`
- `init` 线程 (或进程 1)

迄今为止, 我们已经探讨过 Linux 内核的一些子系统及以操作为中心的数据结构。每章都假定该子系统已经启动并正在运行, 因而主要关注典型的内核子系统的管理及其处理操作。每个子系统在使用前都必须初始化, 这一初始化过程在内核引导时进行, 引导加载程序将内核映像加载到内存并将控制权交给内核后, 才开始进行初始化。

我们根据实际执行的线性次序跟踪内核的初始化过程。首先讨论加电时发生什么, 一直到调用第一个与体系结构无关的函数 `start_kernel()`, 然后关注该函数的运行情况, 直到调用 `/sbin/init` 为止。图 8-1 说明了从系统加电到断电这一过程中所有事件发生的顺序。

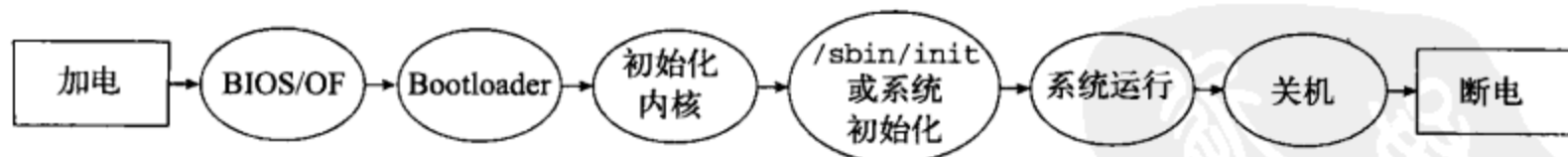


图 8-1 内核的初始状态及其引导过程

首先将讨论 BIOS 和 Open Firmware, 它们分别是 x86 和 PPC 系统加电后最先运行的代码。然后讨论 Linux 系统中常见的引导加载程序, 以及该程序如何加载内核并将执行控制权交给内核。接下来详细讨论内核的初始化过程, 此时要初始化所有子系统。内核初始化的最后一步是由进程 1 调用 `/sbin/init`。某些进程必须在用户登录前运行, `init` 通过将这些进程投入运行来继续系统的初始化。

显然, 内核的部分初始化特性由交叉运行的子系统组成, 因而, 由始至终跟踪给定子系统的整个初始化过程而不被打断, 会非常困难。无论如何, 顺着 Linux 内核引导的线性次序一步一步走下去, 将有利于跟踪内核子系统的实际建立过程, 同时也说明了这个“白手起家”进程的复杂性。

本章将提及前面章节介绍的诸多结构体, 这是因为在本章它们是首次出现并初始化的结构

体。现在，首先要考虑的就是 BIOS 和 Open Firmware。

8.1 BIOS 和 Open Firmware

加电后，处理器首先访问通常位于只读内存中的某一地址。这个只读内存一般是指 Flash ROM（或仅是 Flash），系统上最先运行的代码就存储在这里，这些代码负责启用系统中相应的部分，以便加载内核。

对于 x86 系统而言，这就是系统 BIOS 驻留之处。BIOS（Basic Input Output System，基本输入输出系统）是一段用来引导系统的、与硬件相关的系统初始化代码。在 x86 系统中，引导加载程序和 Linux 依靠 BIOS 将系统带入一个已知的状态。BIOS 的接口是一组统一的名为中断（interrupt）的函数集。内核载入时，Linux 使用这些中断来查询系统的可用资源。BIOS 完成初始化操作后，再从引导设备（参见 8.2 节）复制最初的 512 字节数据到地址 0x7c00 处，并跳转到该地址。虽然在某些安装程序中，BIOS 也通过网络连接来加载操作系统，但从硬盘驱动器加载 Linux 时并不考虑这种情况。加载 Linux 后，BIOS 仍驻留在内存中，其函数也可以被访问，并且能借助中断来调用这些函数。

对 PowerPC 而言，初始化代码的类型与特定 PowerPC 体系结构的出现时间有关。旧式的 IBM 系统使用 PerP（PowerPC Reference Platform，PowerPC 参考平台），当前，更多 IBM 系统使用 CHRP（Common Hardware Reference Platform，通用硬件参考平台）。G4 系统及后续系统“真正新世界”（True New World）以及 OF（Open Firmware）都采用特殊的体系结构实现（关于该处理器和与系统无关的引导固件及其如何使用这些格式，可参见 Open Firmware 的主页 www.openfirmware.org）。

8.2 引导加载程序

引导加载程序（BootLoader）是驻留在计算机引导设备中的程序。第一个引导设备往往是系统中第一个硬盘。完成足够的系统初始化工作（以便支持内存、中断以及加载内核所需的 I/O）后，BIOS（x86 中）或固件（PPC 中）就会调用引导加载程序。成功地将它加载进来以后，内核就开始初始化并配置操作系统。

对 x86 系统而言，BIOS 允许用户为其系统设置引导设备的顺序。典型的引导设备有软盘、光盘和硬盘。格式化磁盘时（例如，使用 fdisk 命令）会创建 MBR（Master Boot Record，主引导记录），该记录存储在引导设备的第一个扇区（0 扇区，0 磁道，0 磁头）。MBR 包含一个小程序和一张四入口点的分区表。引导扇区的末尾在 510 单元处有一个十六进制的标记 0xAA55。表 8-1 给出了 MBR 的组成。

表8-1 MBR的组成

偏移量	长度	作用
0x00	0x1bd	MBR的程序代码
0x1be	0x40	分区表
0x1fe	0x2	十六进制标记或签名

MBR 的分区表保存每个硬盘基本分区的信息。表 8-2 给出了 MBR 分区表的 16 字节表目。

表8-2 MBR的16字节表目

偏移量	长 度	作 用
0x00	1	活动引导分区标志
0x01	3	启动引导分区的柱面/磁头/扇区
0x04	1	分区类型 (Linux使用0x83, 而PPC的PReP使用0x41)
0x05	3	终止引导分区的柱面/磁头/扇区
0x08	4	分区的起始扇区号
0x0c	4	分区长度 (以扇区为单位)

在自检和硬件识别的最后阶段, 系统初始化代码 (固件或 BIOS) 将访问硬件驱动控制器, 以读取。识别出引导驱动器的类型后, 就可以遵循公布的接口 (如 IDE 驱动器) 来访问 0 磁头、0 柱面和 0 扇区。

引导设备定位后, 就将 MBR 复制到内存地址 0x7c00 处并执行。主引导记录开头的一小段程序将自己迁移到别的位置, 并搜索分区表来定位活动引导分区。然后, MBR 将活动引导分区的代码复制到地址 0x7c00 处并开始执行这些代码。由此看来, x86 系统通常引导 DOS, 但是, 活动引导分区可以有一个引导加载程序来依次加载操作系统。下面将讨论 Linux 中最常用的几种引导加载程序。图 8-2 给出了引导时的内存状态。

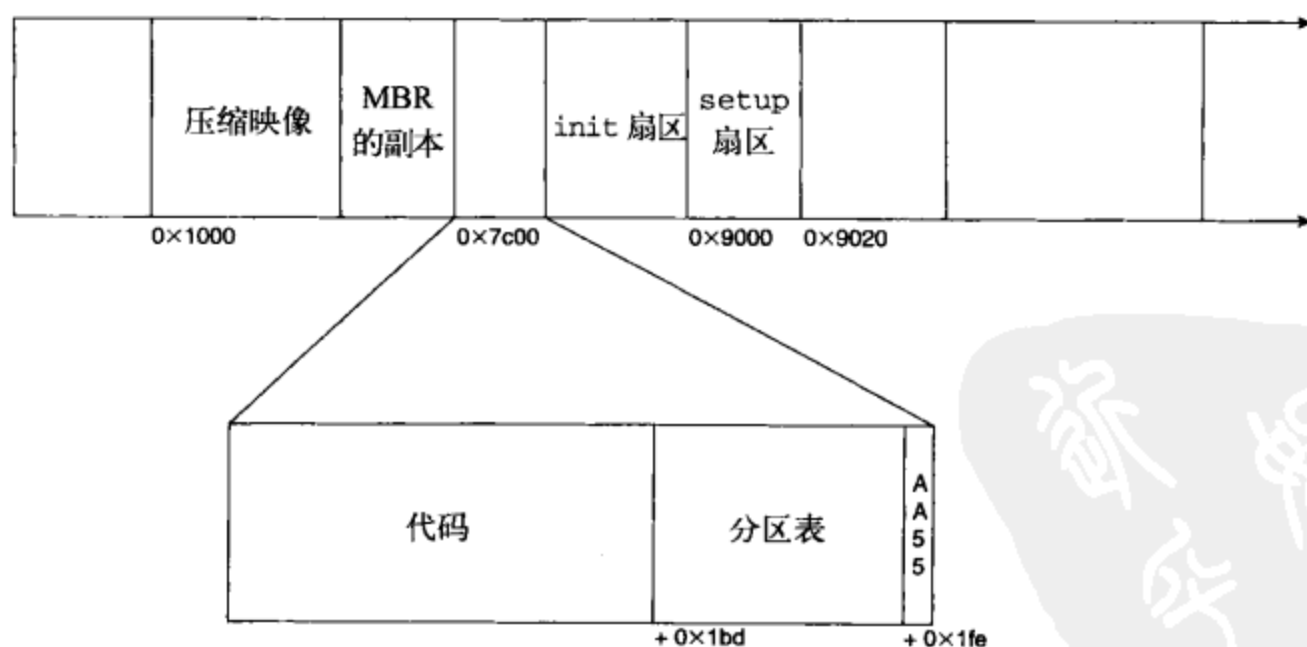


图 8-2 引导时的内存视图

8.2.1 GRUB

GRUB (Grand Unified Bootloader) 是基于 x86 的引导加载程序, 用来加载 Linux。GRUB 2 在设计之初就考虑了移植到 PPC 系统的问题。www.gnu.org/software/grub 上有丰富的相关文档, 包括其发展历史和将来的可扩充性。GRUB 将识别引导驱动器的文件系统, 并且可以通过指定文

件名、驱动器和内核所在的分区来加载内核。GRUB 是一个两阶段引导加载程序^①。BIOS 将调用存储在 MBR 中的第一阶段代码，并由第一阶段将第二阶段的一部分代码装入内存，之后，第二阶段就能借助文件系统来完成自身的加载。GRUB 运行时每一阶段的详细步骤如下。

第一阶段：

- (1) 初始化；
- (2) 检测正在加载的驱动器；
- (3) 加载第一阶段的一个扇区；
- (4) 跳转到第二阶段。

第二阶段：

- (1) 加载第二阶段的剩余代码；
- (2) 跳转到已加载的代码。

通过交互式的命令程序或菜单驱动界面均可访问 GRUB，但使用菜单界面时必须创建一个配置文件。下面是来自加载 Linux 内核的 GRUB 配置文件中的一段代码：

```
-----
/boot/menu.lst
...
title      Kernel 2.6.7, test kernel
root       (hd0,0)
kernel     /boot/bzImage-2.6.7-mytestkernel root=/dev/hda1 ro②
...
-----
```

有如下选项：title，用于保存设置标签；root，用于将当前根设备设置为 hd0 的分区 0；kernel，用于从特定文件中加载主要的内核引导映像。内核入口的其他信息将作为引导参数传给内核。

内核引导的某些方面，例如内核映像加载与解压的位置，将在 Linux 内核代码体与系统结构相关的部分配置。x86 系统中相关内容请查阅 arch/i386/boot/setup.S：

```
-----
arch/i386/boot/setup.S
61 INITSEG = DEF_INITSEG    # 0x9000, we move boot here, out of the way
62 SYSSEG = DEF_SYSSEG      # 0x1000, system loaded at 0x10000 (65536).
63 SETUPSEG = DEF_SETUPSEG  # 0x9020, this is the current segment
-----
```

该配置指定 Linux 将可执行映像加载到线性地址 0x9000 处之后跳转到 0x9020 处。此处，Linux 内核的未压缩部分将压缩部分的代码解压到地址 0x10000 处，并开始初始化内核。

GRUB 基于多重引导规范。本书编写时，Linux 的所有结构并不都是多引导兼容的，但讨论多引导需求依然很有意义。

多重引导规范

多重引导规范 (Multiboot Specification) 描述了任意可能的引导加载程序和任意可能的操作

① 有时，GRUB 也分为 1.5 阶段，但我们仅讨论常用的两级。

② 内核在引导时可以通过内核命令行来接受规范，这是一个描述参数列表的字符串，这些参数用于说明诸如硬件规格、默认值之类的信息。有关 Linux 引导提示的更多信息，可参见 www.tldp.org/HOWTO/BootPrompt-HOWTO.html。

系统之间的接口，它并没有规定引导加载程序该如何做，只是规定与正在装入的操作系统该如何交互。针对当前 x86 体系结构和免费的 32 位操作系统，多重引导规范为引导加载程序提供了一种将配置信息传送到操作系统的标准方式。操作系统的映像可以是任何类型的（ELF 或其他特别的类型），但在其前 8 K 中必须包含多引导头和魔数 0x1BADB002。由于某些操作系统并不将操作所需的所有程序都装入可引导的内核映像，因此多重引导兼容的装载程序还应该提供一种方法，方便操作系统在引导时使用辅助的引导模块和驱动程序，这样做通常可以将引导内核模块化，并将其大小控制在便于管理的规范内。

多重引导规范规定，当引导加载程序激活操作系统时，系统必须运行于特定的 32 位实模式状态，这样操作系统才能在需要时成功地回调 BIOS。最后，引导加载程序必须为操作系统提供填满基本机器数据的数据结构。接下来看看多重引导信息的数据结构：

```
-----
typedef struct multiboot_info
{
    ulong flags;    // indicate following fields
    ulong mem_lower; // if flags[0], amnt of mem < 1M
    ulong mem_upper; // if flags[0], amnt of mem > 1M
    ulong boot_device; // if flags[1], drive, part1,2,3
    ulong cmdline;    // if flags[2], addr of cmd line
    ulong mods_count; // if flags[3], #of boot modules
    ulong mods_addr;  // if flags[3], addr of first
                     // boot module.
    union
    {
        aout_symbol_table_t aout_sym; // if flags[4], symbol table
                                     // from a.out kernel image
        elf_section_header_table_t elf_sec; // if flags[5], header
                                     // from ELF kernel.
    } u;
    ulong mmap_length; // if flags[6], BIOS mem map len
    ulong mmap_addr;  // if flags[6], BIOS map addr
    ulong drives_length; // if flags[7], BIOS drive info structs
    ulong drives_length; // if flags[7], first BIOS drive info
                     // struct.
    ulong config_table // if flags[8], ROM config table
    ulong boot_loader_name // if flags[9], addr of string
    ulong apm_table // if flags[10], addr of APM info table
    ulong vbe_control_info // if flags[11], video mode settings
    ulong vbe_mode_info
    ulong vbe_mode
    ulong vbe_interface_seg
    ulong vbe_interface_off
    ulong vbe_interface_len
};
-----
```

将控制权交给操作系统后，指向该结构体的指针将被存入寄存器 EBX。第一个字段 flags 表明下面哪几个域有效，没有使用的字段必须置为 0。详情可参见 www.gnu.org/software/grub/manual/multiboot/multiboot.html。

8.2.2 LILO

LILO (Linux LOader, Linux 加载程序) 多年来一直用做 x86 中 Linux 的加载程序。它是最早的引导加载程序之一, 用于帮助配置和加载 Linux 内核。LILO 也是一个两阶段的引导加载程序, 从这个角度而言, 它和 GRUB 非常相似, 但 LILO 仅使用配置文件, 并且没有命令行接口。

我们同样从 BIOS 初始化系统并将 MBR (第一阶段) 加载到内存, 然后将控制权交给第一阶段开始探索。LILO 运行时每一阶段的详细步骤如下。

第一阶段:

- (1) 开始执行并显示 “L”;
- (2) 检测磁盘几何信息并显示 “l.”;
- (3) 加载第二阶段的代码。

第二阶段:

- (1) 开始执行并显示 “L”;
- (2) 定位引导数据和操作系统, 并显示 “O.”;
- (3) 确定启动哪个操作系统, 并跳转到该操作系统。

LILO 配置文件中的一段内容如下:

```
-----
/etc/lilo.conf
image=/boot/bzImage-2.6.7-mytestkernel
label=Kernel 2.6.7, my test kernel
root=/dev/hda6
read-only
-----
```

包括以下参数: image, 指明内核所在的路径名; label, 用于描述配置的字符串; root, 指明根文件系统所驻留的分区; read-only, 表明根分区在引导时不可被修改。

以下是 GRUB 与 LILO 之间的区别。

- ❑ LILO 将配置信息存储在 MBR 中。若有任何改动, 必须运行 /sbin/lilo 来更新 MBR。
- ❑ LILO 不能读取不同的文件系统。
- ❑ LILO 没有交互式的命令行接口。

让我们回顾一下使用 LILO 作为引导加载程序时会发生什么。首先, MBR (包含 LILO) 将被复制到 0x7c00 处并开始执行。然后, LILO 开始工作, 它将从硬盘复制/etc/lilo.conf 中引用的内核映像。该映像 (由 build.c 创建) 由 init 扇区 (加载到 0x90000 处)、设置扇区 (加载到 0x90200 处) 和压缩映像 (加载到 0x10000 处) 组成。最后, LILO 将跳转到地址 0x90200 处的标号 start_of_setup 处。

8.2.3 PowerPC 和 Yaboot

Yaboot 是基于新世界 PowerPC 机器的 OF 的引导加载程序。Yaboot 与 LILO 和 GRUB 类似, 它使用配置文件和 ybin 或 ybootconfig 之类的实用程序来设置包含 Yaboot 的引导分区。与 x86 的 BIOS 类似, OF 允许配置引导设备, 但不同的系统, 配置不同。通常, 只要输入 “Command+Option/Alt+o+f.?” 就可以获得 OF 的设置。

Yaboot 引导时的步骤如下。

- (1) OF 调用 Yaboot。
- (2) 找到引导设备和引导路径，并打开引导分区。
- (3) 打开/etc/yaboot.conf 或命令解释器。
- (4) 加载映像或内核以及 initrd。
- (5) 执行映像。

如下所示，Yaboot 中内核加载的这段代码与 LILO 和 GRUB 相似：

```
-----  
yaboot.conf  
label=Linux  
root=/dev/hda11  
sysmap=/boot/System.map  
read-only  
-----
```

与 LILO 中一样，ybin 将 Yaboot 安装到引导分区。更新/修改 Yaboot 的配置后需要重新运行 ybin。

有关 Yaboot 的文档可在 www.penguinppc.org 上找到。

8.3 与体系结构相关的内存初始化

下面将详细介绍 PPC 和 x86 的硬件管理特征。x86 和 PowerPC 在硬件方面都具有支持实寻址和虚寻址的内存管理特征。和其他所有操作系统一样，Linux 内存管理也依赖于底层的硬件结构。本节描述 x86 和 PowerPC 体系结构的硬件初始化。由于内存管理的初始化与硬件息息相关，要理解其初始化过程就必须了解硬件的规格。与体系结构密切相关的天性决定了内存管理是首先进行初始化的子系统之一，并在 `start_kernel()` 执行前就开始被处理。

8.3.1 PowerPC 的硬件内存管理

本节描述 PowerPC 体系结构中硬件支持的地址转换，在 PowerPC 中也被称为“存储控制”。同时也分析 Linux 在从系统加电到内核初始化的过程中如何使用（或因可移植性的缘故而忽略）这些特征。

1. 实寻址模式

无论是嵌入式系统还是高性能系统，所有 PowerPC 的处理器都源于在实模式 (real mode)^① 下复位的硬件。PowerPC 的实寻址模式定义为禁止地址转换的处理器状态。地址转换由 MSR (Machine State Register, 机器状态寄存器) 中的 IR (instruction relocate, 指令重定位) 位和 DR (data relocate, 数据重定位) 位来控制。取指令时，如果 IR 位为 0，那么 EA (effective address, 有效地址) 就是实地址。加载和存储指令时，DR 位在 MSR 中也扮演类似的角色。

图 8-3 解释了 MSR，这是一个 64 位或 32 位寄存器，描述处理器的当前状态。在 32 位 MSR 中，IR 和 DR 分别是第 26 位和第 27 位。

^① 甚至技术上没有实模式的 440 系列处理器也是从“镜像”TLB 开始，该镜像将线性地址映射到物理地址。



图 8-3 PowerPC 的 MSR

Linux 中的地址转换是硬件与软件结构的结合，因此，实模式是初始化内存管理子系统及 Linux 内存管理结构体引导过程的基础。实模式自身的限制说明有必要允许地址转换。实模式仅能寻址已实现的地址宽度，在绝大多数应用中是 64 位或 32 位，其两个主要的局限性如下。

- ❑ 加载/存储操作没有硬件保护。
- ❑ 如果某一地址上没有与总线实际相连的设备，那么任何访问（指令或数据）该地址的操作都可能引发机器自检（也叫做 Checkstop），在绝大多数情况下，这种自检是不可恢复的。

2. 地址转换

实模式寻址没有地址转换。地址转换为虚拟寻址打开了方便之门，此时，在任意给定的实例中，每个可能的地址并不都是可用的。但只要软硬件使用得当，访问时每个可能的地址都能虚拟成可用的地址。

允许地址转换时，PowerPC 体系结构可以使用两种方法来转换 EA，分别是分段地址转换 (Segmented Address Translation) 和块地址转换 (Block Address Translation) (参见图 8-4)。如果这两种方法都可以用于转换该有效地址的话，应该优先使用块地址转换。当 $MSR_{IR} = 1$ 或 $MSR_{DR} = 1$ 或两者均为 1 时，允许地址转换。分段地址转换将虚拟内存分成多个段，每段又分成 4 KB 大小的页，每页对应一页物理内存。块地址转换将内存分成大小为 128 KB 至 256 MB 不等的区域。

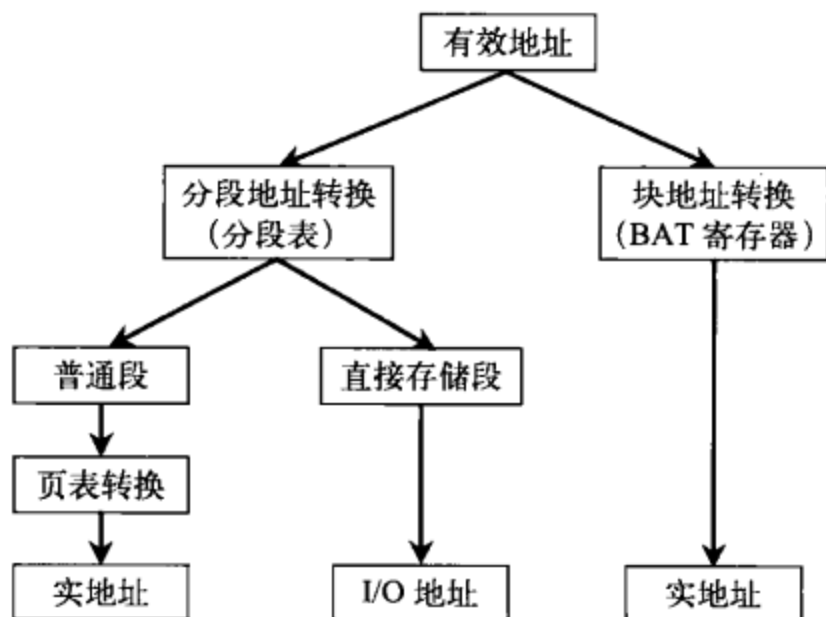


图 8-4 32 位地址转换

内存寻址的术语

提到内存时，只有两种不同的方法和模式：其一是实寻址模式，此时地址每增加 1 都指定了物理内存中的一个基本单元（通常是 1 个字节）；其二是虚拟寻址，此时地址需经过硬件和软件的计算。以下是用于每一寻址模式的几个术语。

- **实寻址 (Real addressing)**: 物理地址, 或总线上的地址。
- **虚拟寻址 (Virtual addressing)**: 有效地址, 或保护地址, 可被转换。

在 PowerPC 中,有效地址空间被认为是虚拟地址空间的子集。

线性、平面、逻辑之类的术语在两种模式下均可使用。

● 分段地址转换中的直接存储段T

下一级地址转换由段寄存器 (Segment Register) 中的二进制位 T 来决定。在 PowerPC 7xx 系列中, 有效地址中的第 0 位~第 3 位用于从 16 个段寄存器 (SR) 中选择一个。图 8-5 描述了段

寄存器。

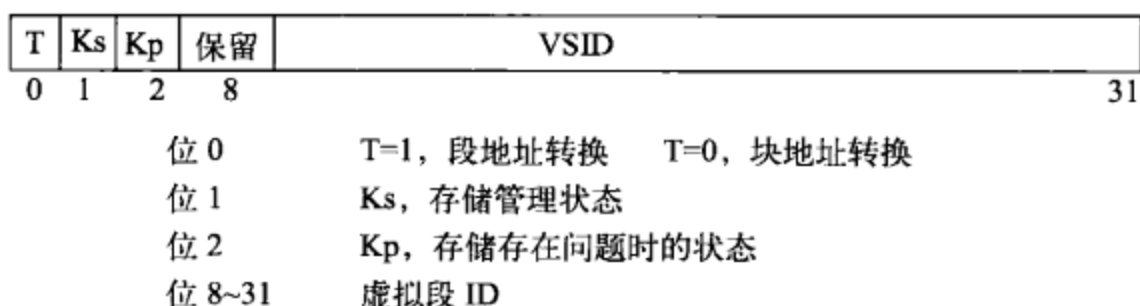


图 8-5 段寄存器

T 置位时, 该段就是 I/O 设备的**直接存储段** (direct store segment), 并且无需引用硬件页表。I/O 地址由权限位、BUID、与控制器相关的字段以及 EA 的第 4 位~第 31 位组成。Linux 没有使用直接存储段。

如果分段地址转换的普通段 T 未置位, 则使用 VSID (virtual segment ID, 虚拟段 ID)。

图 8-6 解释了一个 52 位 VA (virtual address, 虚拟地址) 是如何形成的。该地址由三个部分连接而成, 即有效地址的第 20 位~第 31 位 (给定页中的偏移量)、第 4 位~第 19 位和所选段寄存器 VSID 的第 8 位~第 31 位。VA 的最高 40 位组成了 VPN (virtual page number, 虚拟页号)。PowerPC 体系结构使用散列页表将虚拟页号映射到实际页面号 (即所需页在内存的实际地址)。散列函数通过使用虚拟页号和 SDR1 (Storage Description Register 1, 存储描述寄存器 1) 的值来存储和检索 PTE (Page Table Entry, 页表项)。PTE 是一个 8 字节的数据结构, 它包含内存页面所有必要的属性, 如图 8-7 所示。

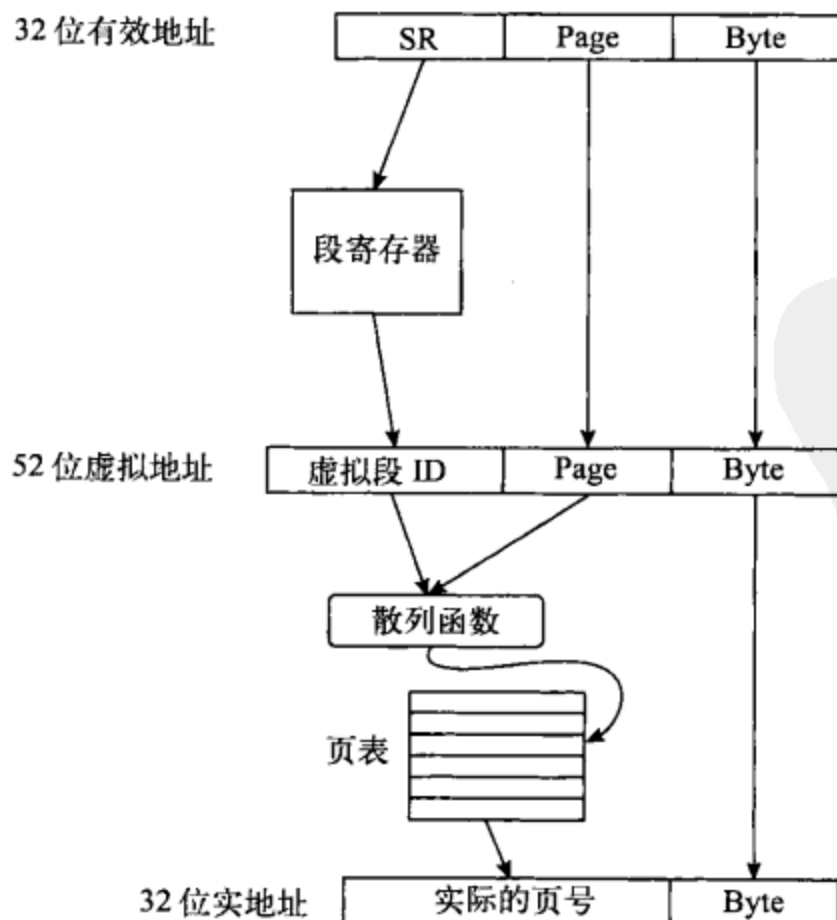


图 8-6 分段地址转换

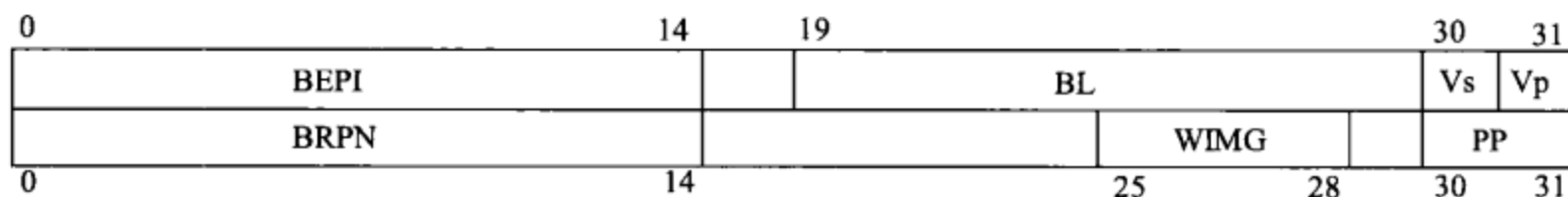


字 0	位 0	V=1, 页表项有效, V=0, 页表项无效
	位 1~24	虚拟段 ID
	位 25	散列函数标识符
	位 26~31	缩写的页索引
字 1	位 0~19	实际的页号
	位 20~22	保留位
	位 23	引用位
	位 24	修改标志位
	位 25~28	WIMG
	位 30~31	用于页保护

图 8-7 页表项

● 块地址转换

顾名思义，BAT (Block Address Translation, 块地址转换) 是允许映射 128 KB~256 MB 不等的连续内存块的寻址机制。在 PowerPC 体系结构中，BAT 寄存器是具有特权的 SPRs (special purpose registers, 专用寄存器)，如图 8-8 所示。



高位寄存器	位 0~14	块的有效页索引
	位 15~18	保留位
	位 25	保留位
	位 19~29	块长度
	位 30	管理状态有效
	位 31	存在问题的状态有效
低位寄存器	位 0~14	块的实际页号
	位 15~24	保留位
	位 25~28	WIMG（参见后面的
	位 29	保留位
	位 30~31	用于对 BAT 区域进行

图 8-8 BAT 寄存器

BAT 寄存器中的信息如何形成实地址可参考图 8-9。通过 PPC 指令^①mtspr 和 mfspr 可

① 并不是所有 PowerPC 的处理器都实现了块地址转换。要注意的是, G4 和 G5 都没有实现块地址转换, 但已经在 4xx-嵌入式处理器中实现了。

以读写 4 个 IBAT (Instruction BAT, 指令块地址转换寄存器) 和 4 个 DBAT (Data BAT, 数据块地址转换寄存器)。

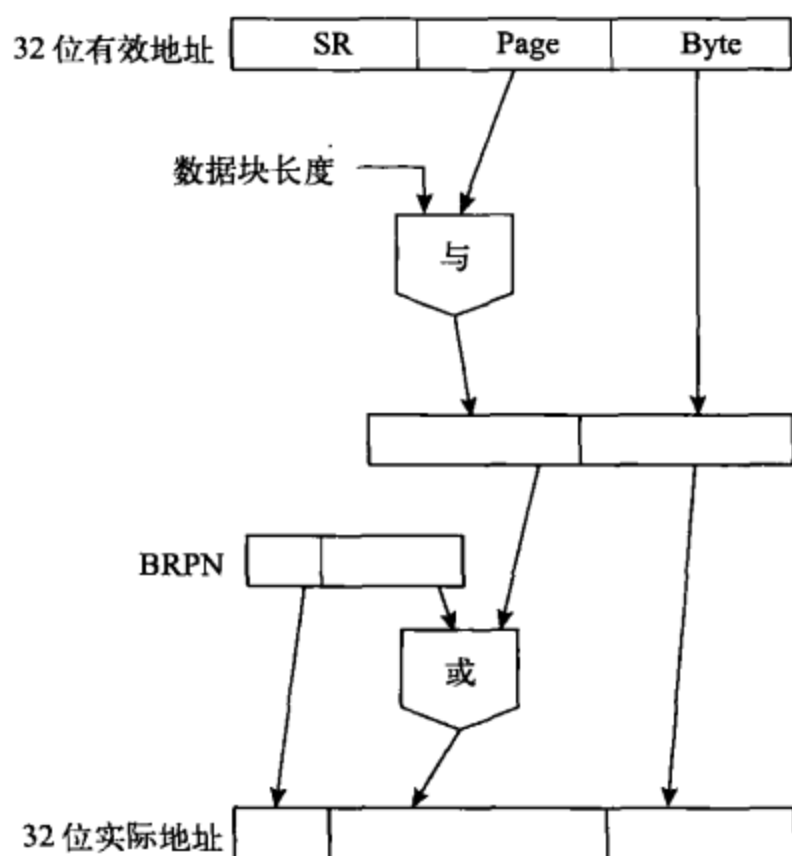


图 8-9 真正的 BAT

● 转换后备缓冲区

转换后备缓冲区 (Translation Lookaside Buffers, TLBs) 简称快表, 可以看做是硬件高速缓存, 它包含内存分页系统的硬件保护部分。TLB 的长度随着 PowerPC 体系结构的变化而有所不同, 它包含最近使用的页表项索引, 因而内存分页软件必须确保 TLB 和页表保持同步。当处理器在散列表^①中找不到某个页面时就会搜索 Linux 的页表, 如果还是没找到, 就会产生正常的缺页异常。有关 Linux 页表和 PPC 散列表之间同步的优化信息, 可参阅 Paul Mackerras 撰写的文档 “Low Level Optimizations in the PowerPC/Linux Kernels” (PowerPC/Linux 内核中的初级优化)。

● 存储访问模式控制 (Storage Access Mode Control)

当地址转换被启用 ($MSR_{IR}=1$, 或 $MSR_{DR}=1$, 或两者均为 1) 并借助分段地址转换或块地址转换来实现时, 存储方式由 4 个控制位来决定: W、I、M 和 G。对分段地址转换而言, 分别是页表项第二个字的第 25 位~第 28 位, 以及数据块地址转换寄存器 (DBAT) 第二个 SPR 的相同位。(G 位保存在指令块地址转换寄存器 IBAT 中。) 此外, 存放在页表项中的引用位和控制位也可用于分段地址转换。R 位和 C 位可由硬件或软件来设置。(有关二进制位 W、I、M、G、R 和 C 的讨论可参阅下面的补充内容)。

^① 散列表并未在所有 PowerPC 处理器中实现。4xx-和 8xx- 嵌入式系统都没有实现散列表, 没有 TLB 时将产生一个硬件和内存分页软件的异常, 之后就会调入所请求的页面。

控制位

二进制位 W、I、M、G、R 和 C 决定处理器如何访问高速缓存和主存。

- ☐ W (Write Through)。若数据存储在高速缓存中并执行了存储操作，当 W=1 时，主存中的备份也必须更新。
- ☐ I (Cache Inhibit)。更新旁路高速缓存并直接访问主存。
- ☐ M (Memory Coherence)。当 M=1 时，强制硬件内存保持一致。
- ☐ G (Guard)。当 G=1 时，禁止使用推测性执行。
- ☐ R (Referenced)。当 R=1 时，页表项被引用。
- ☐ C (Changed)。当 C=1 时，页表项已被修改。

3. Linux 如何使用 PowerPC 的地址转换

让我们来看看 PPC 中哪些影响内存管理的代码。

下面是内核中最先获得控制权的代码。该例程回调 Firmware，并使用 claim() 函数来分配临时空间，之后内核将被解压到适当的位置：

```
-----
arch/ppc/boot/openfirmware/newworldmain.c
40 void boot(int a1, int a2, void *prom)
...
54 claim(initrd_start, RAM_END - initrd_start, 0);
55 printf("initial ramdisk moving 0x%x <- 0x%p (%x bytes)\n\r",
56   initrd_start, (char *)(&__ramdisk_begin), initrd_size);
57 memcpy((char *)initrd_start, (char *)(&__ramdisk_begin), initrd_size);
...
63 /* claim 3MB starting at PROG_START */
64 claim(PROG_START, PROG_SIZE, 0);
65 dst = (void *) PROG_START;
66 if (im[0] == 0x1f && im[1] == 0x8b) {
67 /* claim some memory for scratch space */
68 avail_ram = (char *) claim(0, SCRATCH_SIZE, 0x10);
69 begin_avail = avail_high = avail_ram;
70 end_avail = avail_ram + SCRATCH_SIZE;
71 printf("heap at 0x%p\n", avail_ram);
72 printf("gunzipping (0x%p <- 0x%p:0x%p)...", dst, im, im+len);
73 gunzip(dst, PROG_SIZE, im, &len);
74 printf("done %u bytes\n", len);
75 printf("%u bytes of heap consumed, max in use %u\n",
76   avail_high - begin_avail, heap_max);
...
86 sa = (unsigned long)PROG_START;
87 printf("start address = 0x%x\n", sa);
88
89 (*(kernel_start_t)sa)(a1, a2, prom);
-----
```

第 40 行：该文件的入口点是函数 boot(a1, a2, *prom)。

第 54 行：调用函数 claim() 分配 1 M 以下的内存，ramdisk 的内容也被复制到该内存中。

第 64 行: 调用函数 `claim()` 为内核映像分配从 `0x1_0000` 开始的 3 M 内存。

第 68 行: 调用函数 `claim()` 为 `scratch/heap` 分配从 `0x00` 开始的 8 K 内存。

第 73 行: 内核映像被解压到地址 `0x1_0000` (`PROG_START`) 处。

第 89 行: 使用参数 (`a1`、`a2` 和 `prom`) 跳转到 `0x1_0000` (`(*kernel_start_t)sa`) 处, 此时 `a1` 将值存储在寄存器 `r3` 中 (等于导入 `ramdisk` 的起始地址), `a2` 的值将存储在寄存器 `r4` 中 (等于导入 `ramdisk` 的大小, 如果没有 `ramdisk`, 则大小等于 `0xdeadbeef`), `prom` 的值将存储在寄存器 `r5` 中 (代码存放在系统 ROM 中)。

下列代码块为各种 PowerPC 处理器的硬件内存管理功能做好了准备。RAM 中前 16 M 的内容将被映射到 `0xc0000000` 处:

```
-----
arch/ppc/kernel/head.S
131  __start:
...
150  bl  early_init  in <arch/ppc/kernel/setup.c> (283)
...
170  bl  mmu_off
...
171  RFI: SRR0=>IP, SRR1=>MSR
172  #ifndef CONFIG_POWER4
173  bl  clear_bats
174  bl  flush_tlbs
175
176  bl  initial_bats
177  #if !defined(CONFIG_APUS) && defined(CONFIG_BOOTX_TEXT)
178  bl  setup_disp_bat
179  #endif
180  #else /* CONFIG_POWER4 */
181  bl  reloc_offset
182  bl  initial_mm_power4
183  #endif /* CONFIG_POWER4 */
185  /*
186  * Call setup_cpu for CPU 0 and initialize 6xx Idle
187  */
188  bl  reloc_offset
189  li  r24,0 /* cpu# */
190  bl  call_setup_cpu /* Call setup_cpu for this CPU */
195  #ifdef CONFIG_POWER4
196  bl  reloc_offset
197  bl  init_idle_power4
198  #endif /* CONFIG_POWER4 */
199
210  bl  reloc_offset
211  mr  r26,r3
212  addis r4,r3,KERNELBASE@h /* current address of _start */
213  cmpwi 0,r4,0 /* are we already running at 0? */
214  bne  relocate_kernel
215
...
224  turn_on_mmu:
```

```

225 mfmsr r0
226 ori r0,r0,MSR_DR|MSR_IR
227 mtspr SRR1,r0
228 lis r0,start_here@h
229 ori r0,r0,start_here@l
230 mtspr SRR0,r0
231 SYNC
232 RFI /* enables MMU */

```

第 131 行：这是该代码的入口点。它将获得最小的 mmu 环境设置。（注意，APUS 代表 Amiga 加电系统。）

第 150 行：内核被加载的位置和它被链接的位置可能会有所不同。函数 `early_init` 用于返回当前代码的物理地址。

第 170 行：关闭 PPC 的内存管理单元。如果 IR 和 DR 都被置位则保持其状态，否则禁止重定位。

第 173~176 行：如果没有 power4 和 G5，则清除 BAT 寄存器，刷新 TLBs，并设置 BAT，以便将 RAM 的前 16 M 空间映射到 0xc0000000 处。

要注意的是，整个内核中所使用的各种内核内存标号：

```

-----
arch/ppc/defconfig
CONFIG_KERNEL_START=0xc0000000
-----

```

和

```

-----
include/asm-ppc/page.h
#define PAGE_OFFSET CONFIG_KERNEL_START
#define KERNELBASE PAGE_OFFSET
-----

```

第 181~182 行：通过使用分段来为 power4 和 G5 设置内核内存。

第 188~198 行：`setup_cpu()` 用于初始化内核和用户特征，例如高速缓存配置，以及是否存在 FPU 和 MMU。（注意，直到本书编写时，`init_idle_power4` 还没有实现任何操作。）

第 210 行：根据不同平台将内核重定位到 `KERNELBASE` 或 0x00 处。

第 224~232 行：通过将 MSR 中的 IR 和 DR 置位来启动 MMU（若 MMU 尚未准备就绪）。然后，执行 RFI 指令，跳转到标号 `label_here:` 处（注意，RFI 指令加载包含 SRR1 内容的 MSR，并跳转到 SRR0 所指的地址）。

以下是内核开始运行的代码。它基于命令行来设置系统中的所有内存：

```

-----
arch/ppc/kernel/head.S
1337 start_here:
...
1364 bl machine_init
1365 bl MMU_init
...

```

```

1385 lis r4,2f@h
1386 ori r4,r4,2f@l
1387 tophys(r4,r4)
1388 li r3,MSR_KERNEL & ~(MSR_IR|MSR_DR)
1389 FIX_SRR1(r3,r5)
1390 mtspr SRR0,r4
1391 mtspr SRR1,r3
1392 SYNC
1393 RFI
1394 /* Load up the kernel context */
1395 2: bl load_up_mmu
...
1411 /* Now turn on the MMU for real! */
1412 li r4,MSR_KERNEL
1413 FIX_SRR1(r4,r5)
1414 lis r3,start_kernel@h
1415 ori r3,r3,start_kernel@l
1416 mtspr SRR0,r3
1417 mtspr SRR1,r4
1418 SYNC
1419 RFI

```

第 1337 行：该行是这部分的入口点。

第 1364 行：machine_init()（参见文件 arch/ppc/kernel/setup.c，第 532 行）用于设置与机器相关的信息，例如 NVRAM、L2、CPU 高速缓存的行容量（CPU cache line size，即 CPU 每次与内存交换信息的单位量）以及调试信息，等等。

第 1365 行：MMU_init()（参见文件 arch/ppc/mm/init.c，第 234 行）根据 highmem 和 lowmem 得到总的内存大小，然后初始化内存管理单元的硬件（参见 MMU_init_hw()，第 267 行）、设置散列页表（参见 arch/ppc/mm/hashtable.s）、从 KERNELBASE（参见 mapin_ram()，第 272 行）开始映射所有 RAM 和 I/O（参见 setup_io_mappings()，第 285 行），并初始化上下文管理（参见 mmu_context_init()，第 288 行）。

第 1385 行：关闭 IR 和 DR，设置 SDR1。这样可以保存页表的实地址，还可以知道散列表中有多少位被用于页表索引。

第 1395 行：清除 TLBs，加载 SDR1（即散列表的基址和大小），设置分段，并根据特殊的 PowerPC 平台初始化块地址转换寄存器。

第 1412~1419 行：将 IR、DR 和 RFI 置位，以便启动/init/main.c 中的 start_kernel。注意，当 PowerPC 体系结构发生中断时，ISR（Instruction Address Register，指令地址寄存器）存储中断结束后寄存器返回的地址，该值被保存在 SRR0（Save Restore Register 0）中，相应地，机器状态寄存器被保存在 SRR1（Save Restore Register 1）中。简言之，发生中断时：

□ IAR->SRR0

□ MSR->SRR1

一般在中断例程的末尾将执行指令 RFI，它是上述过程的逆过程，此时用 SRR0 的值恢复 IAR，用 SRR1 恢复 MSR。简言之：

□ SRR0->IAR

□ SRR1->MSR

第 1385~1419 行的代码将通过以下 3 个步骤使用该方法来开启/关闭内存管理功能。

- (1) 在 SRR1 中为 MSR 设置所需的位 (参见图 8-1)。
- (2) 在 SRR0 中设置将要跳转到的地址。
- (3) 执行 RFI 指令。

8.3.2 基于 Intel x86 体系结构的硬件内存管理

加电后, 所有 Intel 处理器都工作在实地址模式下, 这是与早期 Intel 处理器兼容的一种寻址模式。现在, 处理器变得越来越复杂, 但某些遗留下来的代码也一直在使用, 新的处理器仍然必须支持这些代码才能运行。在实地址模式下, 处理器可以使用相同的指令来执行为 8086 和 8088 编写的程序, 更重要的是, 可以使用同一种寻址方式或地址转换 (address translation) 方式。地址转换的最终结果就说明了处理器是如何访问系统内存的。早期的 Intel 处理器有 20 位地址总线, 可以访问大约 64 KB 的内存, 这大大限制了系统中早期代码的大小。在实地址模式下, 线性地址就是物理地址。读者浏览初始化内存管理的代码时, 可以看到近年来处理器硬件方面的更多特性, 以及添加到软件中的更多复杂的数据结构。

setup.S 中执行关于内存初始化的几个重要函数的代码如下:

```
-----
arch/i386/boot/setup.S
307 #define SMAP 0x534d4150
308
309 meme820:
310     xorl    %ebx, %ebx    # continuation counter
311     movw    $E820MAP, %di    # point into the whitelist
312         # so we can have the bios
313         # directly write into it.
314
315     jmpe820:
316     movl    $0x0000e820, %eax    # e820, upper word zeroed
317     movl    $SMAP, %edx    # ascii 'SMAP'
318     movl    $20, %ecx    # size of the e820rec
319     pushw    %ds    # data record.
320     popw    %es
321     int     $0x15    # make the call
322     jc     bail820    # fall to e801 if it fails
323
324     cmpl    $SMAP, %eax    # check the return is 'SMAP'
325     jne     bail820    # fall to e801 if it fails
326
327     ...
333     good820:
334     movb    (E820NR), %al    # up to 32 entries
335     cmpb    $E820MAX, %al
336     jnl     bail820
337
338     incb    (E820NR)
```

```

339  movw  %di, %ax
340  addw  $20, %ax
341  movw  %ax, %di
342  again820:
343  cmpl  $0, %ebx    # check to see if
344  jne   jmpe820     # %ebx is set to EOF
345  bail820:
-----

```

第 307~345 行：考虑代码段时，首先来看一个 BIOS 调用（第 321 行）int 15h，此时 ax=0xe820。该调用返回 BIOS 能够识别的多种不同内存类型的地址和长度。这一简单内存映射表示基本池，可从该基本池获得所有的 Linux 内存页面。进一步研究这些代码可以发现，内存映射可通过 3 种方式获得：0xe820、0xe801 和 0x88。这三种方式都必须与现有的 BIOS 及其平台兼容。

```

-----
arch/i386/boot/setup.S
595  # Now we move the system to its rightful place ... but we check if we have
    a # big-kernel. In that case we *must* not move it ...
597  testb $LOADED_HIGH, %cs:loadflags
598  jz   do_move0    # .. then we have a normal low
599          # loaded zImage
600          # .. or else we have a high
601          # loaded bzImage
602  jmp  end_move    # ... and we skip moving
603
604  do_move0:
605  movw  $0x100, %ax    # start of destination segment
606  movw  %cs, %bp      # aka SETUPSEG
607  subw  $DELTA_INITSEG, %bp    # aka INITSEG
608  movw  %cs:start_sys_seg, %bx    # start of source segment
609  cld
610  do_move:
611  movw  %ax, %es      # destination segment
612  incb  %ah          # instead of add ax, #0x100
613  movw  %bx, %ds      # source segment
614  addw  $0x100, %bx
615  subw  %di, %di
616  subw  %si, %si
617  movw  $0x800, %cx
618  rep
619  movsw
620  cmpw  %bp, %bx      # assume start_sys_seg > 0x200,
621          # so we will perhaps read one
622          # page more than needed, but
623          # never overwrite INITSEG
624          # because destination is a
625          # minimum one page below source
626  jb   do_move
627
628  end_move:
-----

```

第 595~628 行：以下内核映像的代码由 build.c 创建并由 LILO 加载。这些代码由 init 扇区（在地址 0x9000 处）、setup 扇区（在地址 0x9200 处）和压缩映像组成。该映像最初被加载到地址 0x10000 处，如果该映像是 LARGE (>0x7FF) 则留在原地，否则将被迁移到地址 0x1000 处。

```
-----
arch/i386/boot/setup.S
723  # Try enabling A20 through the keyboard controller
724  #endif /* CONFIG_X86_VOYAGER */
725  a20_kbc:
726  call empty_8042
727
728  #ifndef CONFIG_X86_VOYAGER
729  call a20_test      # Just in case the BIOS worked
730  jnz a20_done      # but had a delayed reaction.
731  #endif
732
733  movb $0xD1, %al    # command write
734  outb %al, $0x64
735  call empty_8042
736
737  movb $0xDF, %al    # A20 on
738  outb %al, $0x60
739  call empty_8042
-----
```

如何在 Intel 实地址模式下形成 20 位物理地址

IBM PC 机中最初使用的是 Intel 8088 处理器，仅有 20 位地址线[0...19]。它最多允许系统访问内部 1M 加大约 64 KB 的内存空间 (0~0x10_FFEF)，但物理上（即在总线上）可寻址的最后 64 K 内存实际上就是实存的前 64 K！

在处理器内部，16 位段选择器和 16 位段内偏移量组合形成一个 20 位地址。段选择器左移 4 位再加上段内偏移量，就将地址扩展了 4 位。这些寄存器的和就是总线上的物理地址。

例如：

为了获得最高位地址，可以加载一个值为 0xFFFF 的段寄存器（CS、DS、ES，等等）和一个值为 0xFFFF 的索引寄存器（SI、DI，等等）。在处理器内部，段寄存器左移 4 位并加上偏移量。

0xFFFF 左移 4 位 = 0x0F_FFF0

加上偏移量 + 0x00_FFFF

内部和 = 0x10_FFEF

外部物理地址 = 0x00_FFEF

最终得到的物理地址仍然是一个段选择器（值为 0x0000）和一个偏移量（值为 0xFFEF）的组合（0000: FFEF）。

访问最高地址及其以上的地址将绕回低地址 0xFFEF 处的低端内存。为该处理器编写的某些程序将依赖于这个 20 位的地址回绕行为。Intel 286 及以后的处理器拥有更宽的地址总线，

它与实寻址模式结合起来保持向后兼容 8088 和 8086。实寻址模式并不考虑那些遗留下来并依赖于 20 位地址回绕的软件，但加入了 A20M# 信号引脚来模拟早期处理器的这一特性。该信号可以屏蔽允许再次访问低端内存的 A20 信号。

我们通过一个逻辑门来启用或禁用内存总线的 A20 信号。最初设计这个门时使用的是键盘控制器上额外的 I/O 信号，该信号由 I/O 端口 0x60 和 0x64 来控制。后来又开发了一种使用 I/O 端口 0x92 的“快门 A20” (Fast Gate A20) 方法，并将该端口设计到系统板上。自从所有 x86 处理器都可以在实地址模式下复位以来，引导代码变得更加灵活，也可以使用上述方法来启用特定地址线的 A20 信号。

第 723~739 行：这些代码能够巧妙地转换到早期的 Intel 处理器，但这也给内存管理带来了麻烦。

```
-----
arch/i386/boot/setup.S
790 # set up gdt and idt
791 lidt   idt_48      # load idt with 0,0
792 xorl   %eax, %eax   # Compute gdt_base
793 movw   %ds, %ax     # (Convert %ds:gdt to a linear ptr)
794 shll   $4, %eax
795 addl   $gdt, %eax
796 movl   %eax, (gdt_48+2)
797 lgdt   gdt_48      # load gdt with whatever is
798         # appropriate
...
981 gdt:
982   .fill GDT_ENTRY_BOOT_CS,8,0
983
984   .word 0xFFFF      # 4Gb - (0x100000*0x1000 = 4Gb)
985   .word 0            # base address = 0
986   .word 0x9A00       # code read/exec
987   .word 0x00CF       # granularity = 4096, 386
988         # (+5th nibble of limit)
989
990   .word 0xFFFF      # 4Gb - (0x100000*0x1000 = 4Gb)
991   .word 0            # base address = 0
992   .word 0x9200       # data read/write
993   .word 0x00CF       # granularity = 4096, 386
994         # (+5th nibble of limit)
995 gdt_end:
996   .align 4
997
998   .word 0            # alignment byte
999 idt_48:
1000   .word 0            # idt limit = 0
1001   .word 0, 0         # idt base = 0L
1002
```

```

1003     .word 0      # alignment byte
1004 gdt_48:
1005     .word gdt_end - gdt - 1    # gdt limit
1006     .word 0, 0      # gdt base (filled in later)
-----

```

第 790~797 行：临时 GDT 和 IDT 的结构体和数据都被编译到 setup.S 的末尾，这些表都以最简单的形式来实现。

第 981~1006 行：这些行是临时 GDT 编译到 setup.S 中的值。GDT 有一个代码描述符和一个数据描述符，每个描述符都代表从 0x00 开始的 4 G 内存空间。IDT 被初始化到 0x00 处，稍后给它赋值。

涉及 Intel 平台的内存管理时，进入保护模式是最重要的阶段之一。此时，硬件开始为操作系统构建虚拟地址空间。

保护模式

Intel 的内存管理方式被称为保护模式。“保护”指的是多个相互独立的分段地址空间，它们互不干扰。Intel 内存管理的另一半是内存分页和页面转换。系统程序员可以使用分段和分页的各种组合，但 Linux 使用的是扁平模式 (flat model)，几乎没有用到分段管理。在扁平模式中，每个进程可以使用全部 32 位的地址空间 (4 GB)。

```

-----
arch/i386/boot/setupS
830 movw $1, %ax      # protected mode (PE) bit
831 lmsw %ax          # This is it!
832 jmp flush_instr
833
834 flush_instr:
835 xorw %bx, %bx      # Flag to indicate a boot
836 xorl %esi, %esi    # Pointer to real-mode code
837 movw %cs, %si
838 subw $DELTA_INITSEG, %si
839 shll $4, %esi
-----

```

第 830~831 行：在机器状态字中设置 PE 位将进入保护模式，jmp 指令在保护模式中开始执行。

第 834~839 行：保存指向实模式代码的 32 位指针，用于解压缩和稍后在 startup_32() 中加载内核。

回顾一下，在实寻址模式中，是使用 16 位指令来执行代码，并利用 .code16 汇编指令来编译当前文件，这些指令强制使用实寻址模式，这也就是 *Intel Programmer's Reference* (《Intel 程序员手册》) 中所说的 16 位模块。从 16 位模块跳转到 32 位模块时，Intel 体系结构 (以及汇编程序魔数) 允许在 16 位模块中创建 32 位指令。

下面创建并执行 32 位跳转：

```

-----
arch/i386/boot/setup.S
841 # jump to startup_32 in arch/i386/kernel/head.S
842 #
843 # NOTE: For high loaded big kernels we need a
844 #   jmp 0x100000, __BOOT_CS
845 #
846 # but we haven't yet reloaded the CS register, so the default size
847 # of the target offset still is 16 bit.
848 # However, using an operand prefix (0x66), the CPU will properly
849 # take our 48 bit far pointer. (INTEL 80386 Programmer's Reference
850 # Manual, Mixing 16-bit and 32-bit code, page 16-6)
851
852 .byte 0x66, 0xea    # prefix + jmp-opcode
853 code32: .long 0x1000    # will be set to 0x100000
854          # for big kernels
855 .word __BOOT_CS
-----

```

第 852 行：这一行创建 32 位跳转指令。

执行该跳转后，系统将使用临时 GDT，此时，代码从文件 arch/i386/kernel/head.S 第 57 行的标号 startup_32 处开始就运行在 32 位保护模式下了。

保护模式

迄今为止，我们讨论的都是如何使 Intel 系统准备好，以便设置内存分页。跟踪 head.S 的代码时，可以看到初始化时需要做些什么事情，以及 Linux 如何使用基于 x86 保护模式的分页系统。在 main.c 中，内核启动之前，还需要执行的代码如下所示。有关多种地址转换模式，以及内存初始化和 Intel 处理器的相关设置的完整信息，可参阅 *Intel Architecture Software Developers Manual* (《Intel 体系结构软件开发手册》)，第 III 卷。

```

-----
arch/i386/kernel/head.S
057 ENTRY(startup_32)
058
059 /*
060 * Set segments to known values.
061 */
062 cld
063 lgdt boot_gdt_descr - __PAGE_OFFSET
064 movl $(__BOOT_DS), %eax
065 movl %eax, %ds
066 movl %eax, %es
067 movl %eax, %fs
068 movl %eax, %gs
069
070 /*
071 * Initialize page tables. This creates a PDE and a set of page
072 * tables, which are located immediately beyond _end. The variable
073 * init_pg_tables_end is set up to point to the first "safe" location.
074 */
-----

```



```
085 * Mappings are created both at virtual address 0 (identity mapping)
086 * and PAGE_OFFSET for up to _end+sizeof(page tables)+INIT_MAP_BEYOND_END.
087 *
088 * Warning: don't use %esi or the stack in this code. However, %esp
089 * can be used as a GPR if you really need it...
090 */
091 page_pde_offset = (__PAGE_OFFSET >> 20);
092
093 movl $(pg0 - __PAGE_OFFSET), %edi
094 movl $(swapper_pg_dir - __PAGE_OFFSET), %edx
095 movl $0x007, %eax /* 0x007 = PRESENT+RW+USER */
096 10:
097 leal 0x007(%edi), %ecx /* Create PDE entry */
098 movl %ecx, (%edx) /* Store identity PDE entry */
099 movl %ecx, page_pde_offset(%edx) /* Store kernel PDE entry */
100 addl $4, %edx
101 movl $1024, %ecx
102 11:
103 stosl
104 addl $0x1000, %eax
105 loop 11b
106 /* End condition: we must map up to and including INIT_MAP_BEYOND_END */
107 /* bytes beyond the end of our own page tables; the +0x007 is the attribute bits */
108 leal (INIT_MAP_BEYOND_END+0x007)(%edi), %ebp
109 cmpl %ebp, %eax
110 jb 10b
111 movl %edi, (init_pg_tables_end - __PAGE_OFFSET)
112
113 #ifdef CONFIG_SMP
...
156 3:
157 #endif /* CONFIG_SMP */
158
159 /*
160 * Enable paging
161 */
162 movl $swapper_pg_dir - __PAGE_OFFSET, %eax
163 movl %eax, %cr3 /* set the page table pointer.. */
164 movl %cr0, %eax
165 orl $0x80000000, %eax
166 movl %eax, %cr0 /* ..and set paging (PG) bit */
167 ljmp $__BOOT_CS, $1f /* Clear prefetch and normalize %eip */
168 1:
169 /* Set up the stack pointer */
170 lss stack_start, %esp
...
177 pushl $0
178 popfl
179
```

```

180 #ifdef CONFIG_SMP
181     andl %ebx,%ebx
182     jz 1f      /* Initial CPU cleans BSS */
183     jmp checkCPUtype
184 1:
185 #endif /* CONFIG_SMP */
186
187 /*
188  * start system 32-bit setup. We need to re-do some of the things done
189  * in 16-bit mode for the "real" operations.
190  */
191     call setup_idt
192
193 *
194 * Copy bootup parameters out of the way.
195 * Note: %esi still has the pointer to the real-mode data.
196 */
197     movl $boot_params,%edi
198     movl $(PARAM_SIZE/4),%ecx
199     cld
200     rep
201     movsl
202     movl boot_params+NEW_CL_POINTER,%esi
203     andl %esi,%esi
204     jnz 2f     # New command line protocol
205     cmpw $(OLD_CL_MAGIC),OLD_CL_MAGIC_ADDR
206     jne 1f
207     movzwl OLD_CL_OFFSET,%esi
208     addl $(OLD_CL_BASE_ADDR),%esi
209 2:
210     movl $saved_command_line,%edi
211     movl $(COMMAND_LINE_SIZE/4),%ecx
212     rep
213     movsl
214 1:
215     checkCPUtype:
216     ...
279     lgdt cpu_gdt_descr
280     lidt idt_descr
281     ...
303     call start_kernel
-----

```

第 57 行：这是内核代码进入 32 位保护模式的入口点。当前这些代码使用临时 GDT。

第 63 行：该代码用引导时 GDT 的基地址初始化 GDTR。这个引导时的 GDT 与 setup.S 中所用的临时 GDT 相同（从地址 0x00000000 处开始的 4 GB 代码和数据），并仅由该引导代码使用。

第 64~68 行：__BOOT_DS 的值为（其取值请参见/include/asm-i386/segment.h），用于初始化剩余的段寄存器。该值指向 GDT 末尾的第 24 个选择器（从 0 开始编号），稍后将设置该寄

寄存器。

第 91~111 行：在 `swapper_pg_dir` 中创建一个页目录项 (PDE)，该页目录项引用页表 (`pg0`)，`pg0` 一开始的内容全部为 0，创建第一个目录项之后将复制 `PAGE_OFFSET` (内核内存) 的目录项。

第 113~157 行：该代码块初始化第二个 (非引导) 处理器的页表。我们主要讨论的是引导处理器。

第 162~164 行：寄存器 `cr3` 是 x86 中硬件分页的入口点。该寄存器被初始化为指向页目录 (Page Directory) 的基址，本例中是 `swapper_pg_dir`。

第 165~168 行：设置引导处理器中寄存器 `cr0` 的 PG (分页) 位。PG 位可启用 x86 体系结构的分页机制。进入或退出分页模式时，需要改变 PG 位，以确保处理器中所有指令都串行执行，建议使用跳转指令来实现 (见第 167 行)。

第 170 行：将栈初始化为数据段的起始位置 (可参阅第 401~403 行)。

第 177~178 行：寄存器 `eflags` 是读/写系统寄存器，它包含中断状态、模式和许可权。清除该寄存器可以使用如下方法：将 0 压入栈，然后使用 `popfl` 指令直接将它弹出到该寄存器。

第 180~185 行：通用寄存器 `ebx` 存储标志信息，标明运行该代码的处理器究竟是不是引导处理器。我们将它当做引导处理器而跟踪这些代码，因而 `ebx` 被清 0，并跳转到调用 `setup_idt` 的地址处。

第 191 行：`setup_idt` 例程初始化 IDT (Interrupt Descriptor Table, 中断描述符表)，它的每个表项都指向一个虚拟句柄。在第 7 章中已经介绍了 IDT，表中是一些函数 (或处理程序)，当处理器需要立即执行紧急代码时就调用这些函数。

第 197~214 行：用户可以在引导时传递某些参数给 Linux，这些参数存储在此处，稍后将会用到。

第 215~303 行：这些代码为 x86 处理器做了大量乏味但必须的工作，即版本检测和一些次要的初始化。借助于 `cupid` 指令 (或者不用该指令) 可以设置寄存器 `eflags` 和寄存器 `cr0` 的某些位。`cr0` 中值得注意的是位 4，它代表扩展类型 (ET)。该位表明支持老式 x86 处理器中的数学协处理器指令。这块代码中最重要的是第 279~280 行，它们将 IDT 和 GDT (借助 `lidt` 和 `lgdt` 指令) 加载到寄存器 `idtr` 和 `gdtr`。最后，第 303 行，跳转到例程 `start_kernel()` 处。

系统使用 `head.S` 中的代码可以将逻辑地址映射到线性地址，最后映射到物理地址 (参见图 8-10)。从逻辑地址开始，段选择器 (CS、DS、ES 等) 引用 GDT 中的一个描述符，而偏移量就是找到的扁平地址。描述符和偏移量组合，就形成了逻辑地址。

代码遍查时，读者可以看到如何创建页目录 (`swapper_pg_dir`) 和页表 (`pg0`)，以及寄存器 `cr3` 如何初始化为指向页目录。如前所述，处理器根据寄存器 `cr3` 的设置就知道要到何处寻找内存分页的部件，并设置寄存器 `cr0` (PG 位)，以通知处理器可以开始使用这些部件了。逻辑地址的第 22~31 位表示 PDE，第 12~21 位表示 PTE，第 0~11 位表示物理页内的偏移量 (本例中是 4 KB)。

现在，该系统使用临时分页系统可以定制 8 MB 内存。下一步就是调用 `init/main.c` 中的函数 `start_kernel()`。

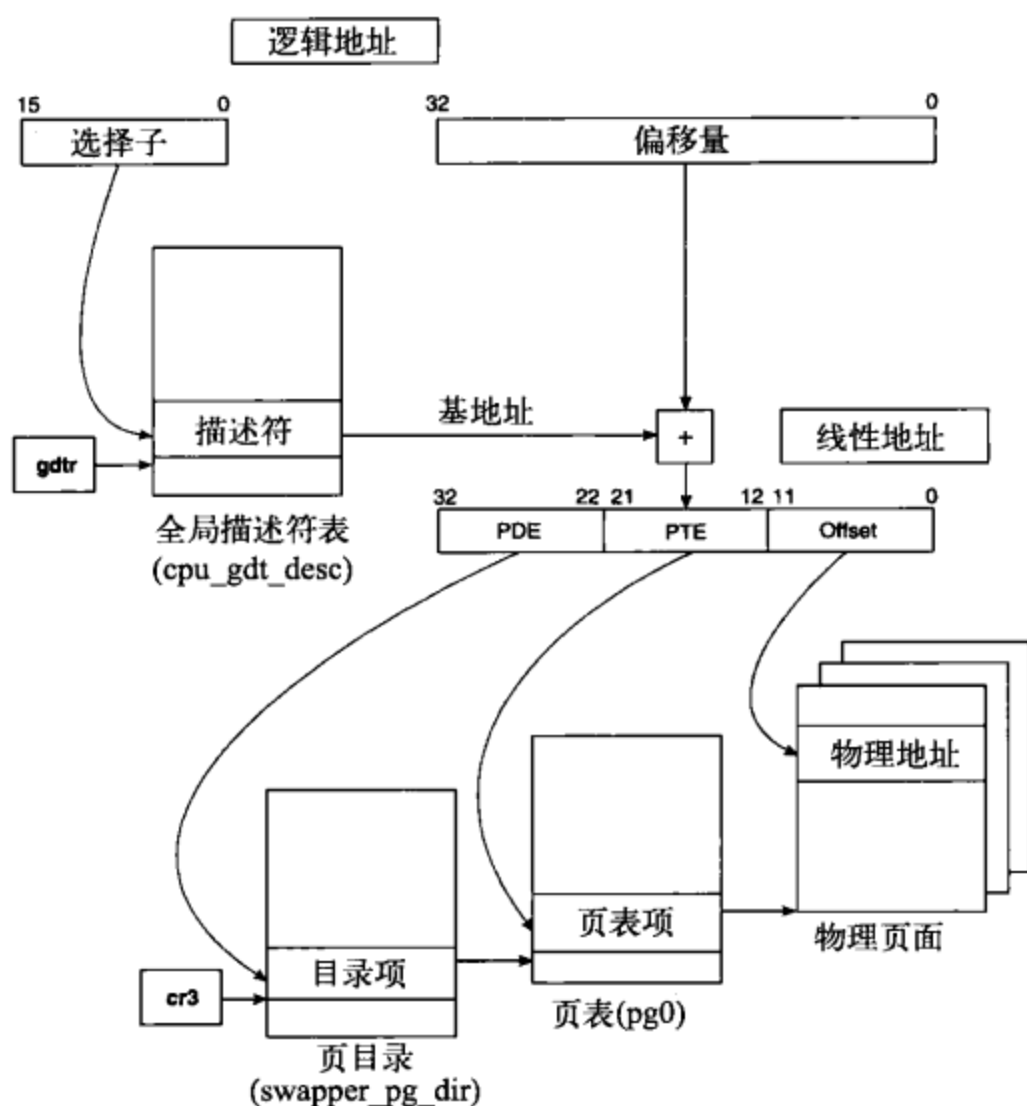


图 8-10 引导时的内存分页

8.3.3 PowerPC 和 x86 的代码汇集

注意：现在 PowerPC 和 x86 的代码都集中在 `init/main.c` 的 `start_kernel()` 中。该例程位于与体系结构无关的代码段中，将调用特定体系结构的例程来完成内存初始化。

该文件中调用的第一个函数就是 `arch/i386/kernel/setup.c` 中的 `setup_arch()`，再调用 `arch/i386/mm/init.c` 中的 `paging_init()`，之后 `paging_init()` 调用同一文件中的 `pagetable_init()`。系统中其余的内存在此处被分配出去，以生成最后的页表。

在 PowerPC 的世界里，很多事情都已经完成了。接下来，`arch/ppc/kernel/setup.c` 中的 `setup_arch()` 将调用 `arch/ppc/mm/init.c` 内的 `paging_init()`。值得注意的是，PPC 的 `paging_init()` 会将所有页面都设置到 DMA 区。

8.4 原始的 RAM 盘

LILO、GRUB 和 Yaboot 都可以加载原始的 RAM 盘 (`initrd`)。在加载并初始化最终的根文件系统之前，`initrd` 扮演根文件系统的角色。我们也把最终根文件系统的加载称为根的旋转。

该初始化步骤允许 Linux 一开始就加载某些预编译模块，然后动态加载其他模块和 `initrd` 的驱动程序。`initrd` 与引导加载程序的主要区别是，它在第二阶段时加载一个最小的内核和

RAM 盘。内核利用 RAM 盘来初始化并挂载最终的根文件系统，然后删除 initrd。

initrd 支持如下操作：

- 引导时配置内核；
- 保持一个小的通用内核；
- 有一个用于若干硬件配置的内核；

前面引用的章节是使用 Yaboot、GRUB 和 LILO 装载 Linux 时最常见的。每个引导加载程序都有丰富的用于配置文件的命令集。对于定制的或特殊的引导过程而言，可以在网上搜索到很多与 GRUB 和 LILO 配置文件相关的信息。

现在，我们已经讨论过如何加载内核以及如何启动内存的初始化过程，接下来就要探讨内核的初始化过程了。

8.5 开始：start_kernel()

我们从函数 start_kernel()（见 init/main.c）开始讨论，这是最先调用的与体系结构无关的代码。

跳转到 start_kernel()时，会执行进程 0，也就是通常所说的根用户线程（root thread）。进程 0 孕育了进程 1，也就是 init 进程，然后进程 0 就变成 CPU 的空闲进程。在调用/sbin/init 时，系统中仅有这两个进程在运行：

```
-----
init/main.c
396 asmlinkage void __init start_kernel(void)
397 {
398     char * command_line;
399     extern char saved_command_line[];
400     extern struct kernel_param __start__param[], __stop__param[];
401     ...
405     lock_kernel();
406     page_address_init();
407     printk(linux_banner);
408     setup_arch(&command_line);
409     setup_per_cpu_areas();
410     ...
415     smp_prepare_boot_cpu();
416     ...
422     sched_init();
423
424     build_all_zonelists();
425     page_alloc_init();
426     printk("Kernel command line: %s\n", saved_command_line);
427     parse_args("Booting kernel", command_line, __start__param,
428         __stop__param - __start__param,
429         &unknown_bootoption);
430     sort_main_extable();
431     trap_init();
432     rcu_init();
433     init_IRQ();
434     pidhash_init();
```

```

435  init_timers();
436  softirq_init();
437  time_init();
...
444  console_init();
445  if (panic_later)
446    panic(panic_later, panic_param);
447  profile_init();
448  local_irq_enable();
449  #ifdef CONFIG_BLK_DEV_INITRD
450    if (initrd_start && !initrd_below_start_ok &&
451        initrd_start < min_low_pfn << PAGE_SHIFT) {
452      printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
453        "disabling it.\n", initrd_start, min_low_pfn << PAGE_SHIFT);
454      initrd_start = 0;
455    }
456  #endif
457  mem_init();
458  kmem_cache_init();
459  if (late_time_init)
460    late_time_init();
461  calibrate_delay();
462  pidmap_init();
463  pgtable_cache_init();
464  prio_tree_init();
465  anon_vma_init();
466  #ifdef CONFIG_X86
467    if (efi_enabled)
468      efi_enter_virtual_mode();
469  #endif
470  fork_init(num_physpages);
471  proc_caches_init();
472  buffer_init();
473  unnamed_dev_init();
474  security_scaffolding_startup();
475  vfs_caches_init(num_physpages);
476  radix_tree_init();
477  signals_init();
478  /* rootfs populating might need page-writeback */
479  page_writeback_init();
480  #ifdef CONFIG_PROC_FS
481    proc_root_init();
482  #endif
483  check_bugs();
...
490  init_idle(current, smp_processor_id());
...
493  rest_init();
494  }

```

8.5.1 调用 lock_kernel()

第 405 行: 在 Linux 2.6 内核中, 默认的配置支持抢占式内核。抢占式内核意味着内核本身

可以被优先级更高的任务中断，如硬件中断，并由高优先级的任务获得控制权。因此，内核必须保存足够多的状态信息，以便高优先级的任务完成后能够返回并执行原来的任务。

早期版本的 Linux 利用 BKL (Big Kernel Lock, 大内核锁) 实现了内核抢占和 SMP (对称多处理器) 锁定。在后来的版本中，Linux 将抢占抽象成各种调用，比如 `preempt_disable()`。BKL 至今仍用于初始化过程，它是一个递归的自旋锁，可以被给定的 CPU 多次运行。使用 BKL 的副作用就是它禁止内核抢占，这对初始化过程非常不利。

锁住内核可以防止它被任何其他任务中断或抢占，Linux 是使用 BKL 来实现这一功能的。当内核被锁住时，其他任何进程都不能被执行，这与任何时候都可以中断的抢占式内核刚好相反。在 Linux 2.6 内核中，使用 BKL 在启动时锁住内核，并初始化各种内核对象而不用担心被中断。第 493 行的 `rest_init()` 函数将内核解锁。因此，`start_kernel()` 的所有工作都将在内核被锁住时完成。接下来让我们来看看 `lock_kernel()` 函数会做些什么：

```
-----
include/linux/smp_lock.h
42 static inline void lock_kernel(void)
43 {
44     int depth = current->lock_depth+1;
45     if (likely(!depth))
46         get_kernel_lock();
47     current->lock_depth = depth;
48 }
-----
```

第 44~48 行：init 任务有一个特殊的 `lock_depth`，其值为 -1，它确保在多处理器系统中，不同的 CPU 不会试图同时抢夺内核锁。由于仅有一个 CPU 运行 init 任务，且只有 init 的 `depth` 是 0（其他所有任务的 `depth` 都大于 0），因此，只有该任务可以抢夺大内核锁。函数 `unlock_kernel()` 中也采用了类似的技巧，可以在该函数中测试 (`--current->lock_depth<0`)。现在，让我们来看看函数 `get_kernel_lock()` 会做些什么：

```
-----
include/linux/smp_lock.h
10 extern spinlock_t kernel_flag;
11
12 #define kernel_locked()    (current->lock_depth >= 0)
13
14 #define get_kernel_lock()  spin_lock(&kernel_flag)
15 #define put_kernel_lock()  spin_unlock(&kernel_flag)
...
59 #define lock_kernel()      do { } while(0)
60 #define unlock_kernel()     do { } while(0)
61 #define release_kernel_lock(task) do { } while(0)
62 #define reacquire_kernel_lock(task) do { } while(0)
63 #define kernel_locked()    1
-----
```

第 10~15 行：这些宏描述那些使用标准自旋锁例程的大内核锁。在多处理器系统中，可能会有两个 CPU 试图访问同一个数据结构，在第 7 章阐述的自旋锁可以预防这种争夺。

第 59~63 行: 当内核不能被抢占且不会运行于多个 CPU 上时, 由于不会发生任何中断, 因而 lock_kernel() 什么都不做。

现在, 内核已经抓住了 BKL, 直到 start_kernel() 结束时才会释放, 因此, 以下所有命令都不能被抢占。

8.5.2 调用 page_address_init()

第 406 行: 在与体系结构相关部分的代码中, page_address_init() 是内存子系统初始化时最先调用的函数, 其定义根据编译时 3 个不同参数的值而定。前两个参数通过将函数体定义为 do{}while(0), 使得 page_address_init() 停下来什么都不做, 如下列代码所示。第三个参数将在此处详细探讨。让我们来看看这些不同的定义, 并讨论它们在什么情况下启用:

```
-----
include/linux/mm.h
376 #if defined(WANT_PAGE_VIRTUAL)
382 #define page_address_init() do { } while(0)

385 #if defined(HASHED_PAGE_VIRTUAL)
388 void page_address_init(void);

391 #if !defined(HASHED_PAGE_VIRTUAL) && !defined(WANT_PAGE_VIRTUAL)
394 #define page_address_init() do { } while(0)
-----
```

当系统允许直接映射内存时, 就设置 WANT_PAGE_VIRTUAL 的 #define, 此时仅计算出其虚拟地址就能够访问对应的存储单元了。万一所有 RAM 都无法映射到内核地址空间 (配置了高端内存 himem 后会经常发生这种情况), 就需要更多相关途径来获得内存地址。因此, 页面寻址的初始化只有在设置了 HASHED_PAGE_VIRTUAL 后才定义。

现在, 来考虑一下内核在何处需被告知使用了 HASHED_PAGE_VIRTUAL, 并在何处需要初始化内核正在使用的虚拟内存。记住, 这只有当配置了 himem 时才会发生, 就是说, 内核可以访问的 RAM 大小远比能被内核地址空间映射的 RAM 大小大得多 (通常是 4 GB)。

定义以下函数的过程中将会引入并反复访问各种内核对象。表 8-3 给出了分析函数 page_address_init() 引入到的内核对象。

表8-3 调用函数page_address_init()时引入的内核对象

对 象 名	描 述
page_address_map	结构体
page_address_slot	结构体
page_address_pool	全局变量
page_address_maps	全局变量
page_address_htable	全局变量

```
-----
mm/highmem.c
510 static struct page_address_slot {
511     struct list_head lh;
```

```

512 spinlock_t lock;
513 } ____cacheline_aligned_in_smp page_address_htable[1<<PA_HASH_ORDER];
...
591 static struct page_address_map page_address_maps[LAST_PKMAP];
592
593 void __init page_address_init(void)
594 {
595     int i;
596
597     INIT_LIST_HEAD(&page_address_pool);
598     for (i = 0; i < ARRAY_SIZE(page_address_maps); i++)
599         list_add(&page_address_maps[i].list, &page_address_pool);
600     for (i = 0; i < ARRAY_SIZE(page_address_htable); i++) {
601         INIT_LIST_HEAD(&page_address_htable[i].lh);
602         spin_lock_init(&page_address_htable[i].lock);
603     }
604     spin_lock_init(&pool_lock);
605 }

```

第 597 行：本行旨在初始化全局变量 `page_address_pool`，这是一个 `list_head` 类型的结构体，指向从 `page_address_maps`（第 591 行）分配的空闲页链表。图 8-11 解释了 `page_address_pool`。

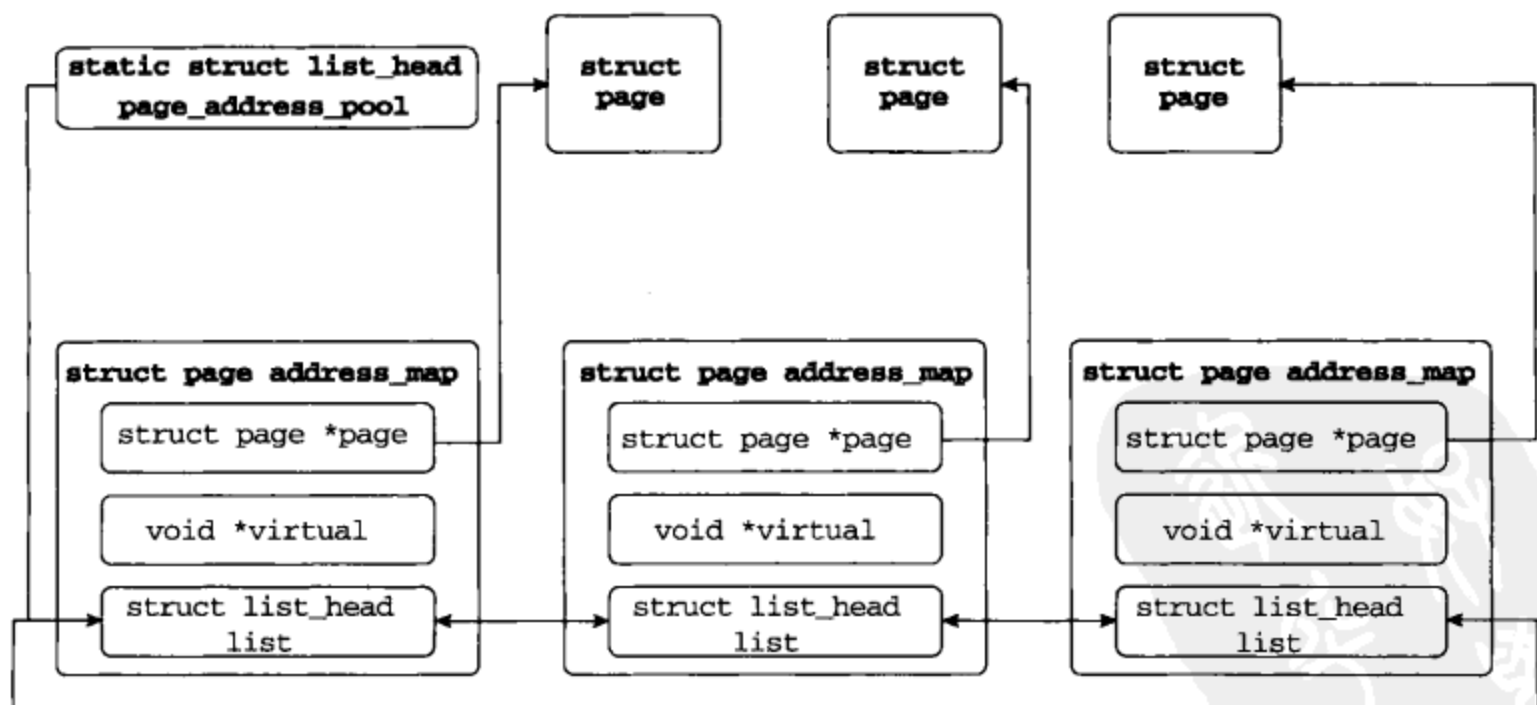


图 8-11 围绕页地址映射池的数据结构

第 598~599 行：将 `page_address_map` 的每个页链表加入到以 `page_address_pool` 为表头的双向链表中。稍后将详细讨论结构体 `page_address_map`。

第 600~603 行：初始化每个页地址散列表的 `list_head` 和自旋锁。散列到同一地址的元素形成的链表将存放在变量 `page_address_htable` 中。图 8-12 描述了页地址散列表。

第 604 行：初始化 `page_address_pool` 的自旋锁。

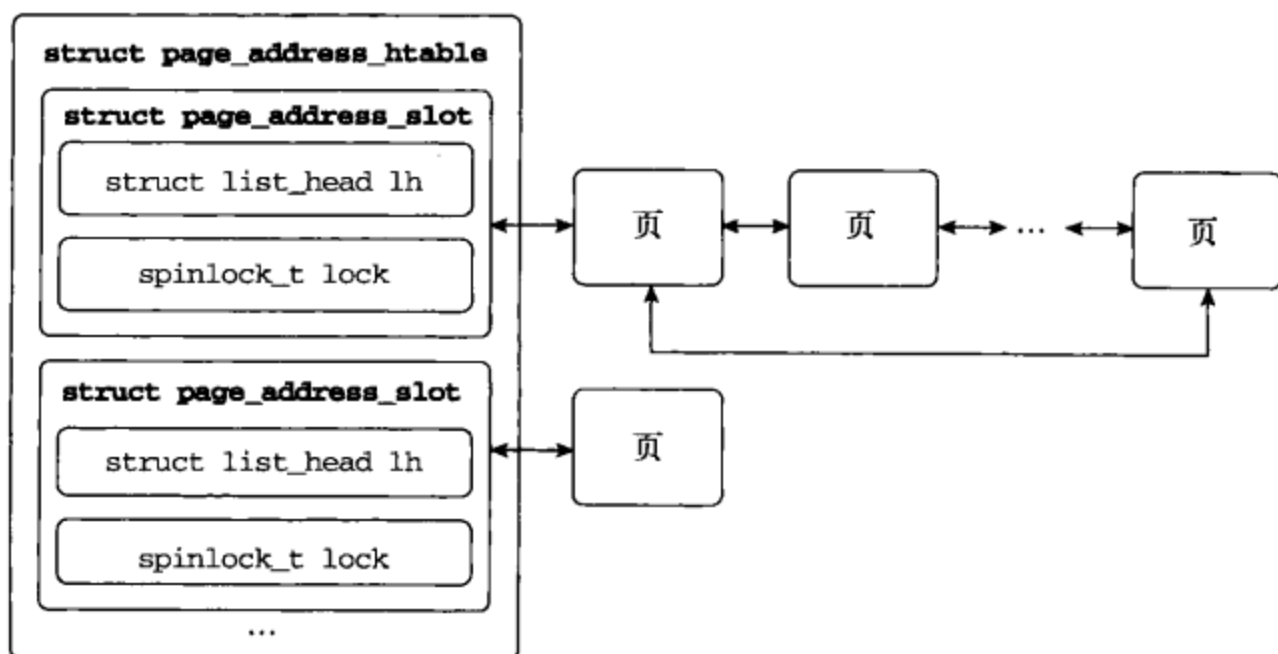


图 8-12 页地址散列表

下面, 分析一下结构体 `page_address_map`, 以便更好地理解刚才初始化过的列表。该结构体旨在保持页面及其虚拟地址之间的联系, 如果页面与其虚拟地址之间是一个线性关系的话就太浪费了。仅当使用散列寻址时该结构体才有必要存在:

```
-----
mm/highmem.c
490 struct page_address_map {
491     struct page *page;
492     void *virtual;
493     struct list_head list;
494 };
-----
```

可以看到, 该对象用于保存指向与该页相关的页结构的指针、指向虚拟地址的指针, 以及 `list_head` 结构体, 以便保持它在页面地址所形成的双向链表中的位置。

8.5.3 调用 `printk(linux_banner)`

第 407 行: 该调用负责 Linux 内核第一个控制台的输出。此处引入全局变量 `linux_banner`:

```
-----
init/version.c
31 const char *linux_banner =
32     "Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n";
-----
```

如上例所示, 文件 `version.c` 中定义了 `linux_banner`, 该字符串将为用户提供内核版本, 用来编译该内核的 gcc 版本, 以及发布版本等参考信息。

8.5.4 调用 `setup_arch`

第 408 行: `arch/i386/kernel/setup.c` 中的函数 `setup_arch()` 被转换成 `__init` 类型 (有关 `__init`

的描述参见第2章), 并仅在系统初始化时运行一次。函数 `setup_arch()` 接收一个指向引导时输入的任何 Linux 命令行数据的指针, 并初始化许多特定于体系结构的子系统, 例如内存、I/O、处理器和控制台:

```
-----
arch/i386/kernel/setup.c
1083 void __init setup_arch(char **cmdline_p)
1084 {
1085     unsigned long max_low_pfn;
1086
1087     memcpy(&boot_cpu_data, &new_cpu_data, sizeof(new_cpu_data));
1088     pre_setup_arch_hook();
1089     early_cpu_init();
1090
1091     /*
1092     * FIXME: This isn't an official loader_type right
1093     * now but does currently work with elilo.
1094     * If we were configured as an EFI kernel, check to make
1095     * sure that we were loaded correctly from elilo and that
1096     * the system table is valid. If not, then initialize normally.
1097     */
1098     #ifdef CONFIG_EFI
1099     if ((LOADER_TYPE == 0x50) && EFI_SYSTAB)
1100         efi_enabled = 1;
1101     #endif
1102
1103     ROOT_DEV = old_decode_dev(ORIG_ROOT_DEV);
1104     drive_info = DRIVE_INFO;
1105     screen_info = SCREEN_INFO;
1106     edid_info = EDID_INFO;
1107     apm_info.bios = APM_BIOS_INFO;
1108     ist_info = IST_INFO;
1109     saved_videomode = VIDEO_MODE;
1110     if( SYS_DESC_TABLE.length != 0 ) {
1111         MCA_bus = SYS_DESC_TABLE.table[3] & 0x2;
1112         machine_id = SYS_DESC_TABLE.table[0];
1113         machine_submodel_id = SYS_DESC_TABLE.table[1];
1114         BIOS_revision = SYS_DESC_TABLE.table[2];
1115     }
1116     aux_device_present = AUX_DEVICE_INFO;
1117
1118     #ifdef CONFIG_BLK_DEV_RAM
1119     rd_image_start = RAMDISK_FLAGS & RAMDISK_IMAGE_START_MASK;
1120     rd_prompt = ((RAMDISK_FLAGS & RAMDISK_PROMPT_FLAG) != 0);
1121     rd_doload = ((RAMDISK_FLAGS & RAMDISK_LOAD_FLAG) != 0);
1122     #endif
1123     ARCH_SETUP
1124     if (efi_enabled)
1125         efi_init();
1126     else
1127         setup_memory_region();
1128
1129     copy_edd();
1130
```

```
1131 if (!MOUNT_ROOT_RDONLY)
1132     root_mountflags &= ~MS_RDONLY;
1133 init_mm.start_code = (unsigned long) _text;
1134 init_mm.end_code = (unsigned long) _etext;
1135 init_mm.end_data = (unsigned long) _edata;
1136 init_mm.brk = init_pg_tables_end + PAGE_OFFSET;
1137
1138 code_resource.start = virt_to_phys(_text);
1139 code_resource.end = virt_to_phys(_etext)-1;
1140 data_resource.start = virt_to_phys(_etext);
1141 data_resource.end = virt_to_phys(_edata)-1;
1142
1143 parse_cmdline_early(cmdline_p);
1144
1145 max_low_pfn = setup_memory();
1146
1147 /*
1148  * NOTE: before this point _nobody_ is allowed to allocate
1149  * any memory using the bootmem allocator.
1150  */
1151
1152 #ifdef CONFIG_SMP
1153     smp_alloc_memory(); /* AP processor realmode stacks in low memory*/
1154 #endif
1155 paging_init();
1156
1157 #ifdef CONFIG_EARLY_PRINTK
1158 {
1159     char *s = strstr(*cmdline_p, "earlyprintk=");
1160     if (s) {
1161         extern void setup_early_printk(char *);
1162
1163         setup_early_printk(s);
1164         printk("early console enabled\n");
1165     }
1166 }
1167 #endif
1168 ...
1170 dmi_scan_machine();
1171
1172 #ifdef CONFIG_X86_GENERICARCH
1173     generic_apic_probe(*cmdline_p);
1174 #endif
1175 if (efi_enabled)
1176     efi_map_memmap();
1177
1178 /*
1179  * Parse the ACPI tables for possible boot-time SMP configuration.
1180  */
1181 acpi_boot_init();
1182
1183 #ifdef CONFIG_X86_LOCAL_APIC
1184     if (smp_found_config)
1185         get_smp_config();
1186 #endif
```



```

1187
1188 register_memory(max_low_pfn);
1188
1190 #ifdef CONFIG_VT
1191 #if defined(CONFIG_VGA_CONSOLE)
1192     if (!efi_enabled || (efi_mem_type(0xa0000) != EFI_CONVENTIONAL_MEMORY))
1193         conswitchp = &vga_con;
1194 #elif defined(CONFIG_DUMMY_CONSOLE)
1195     conswitchp = &dummy_con;
1196 #endif
1197 #endif
1198 }

```

第 1087 行：获得 `boot_cpu_data` 的值，这是指向 `cpuinfo_x86` 结构体的指针，引导时给该结构体赋值。PPC 系统中与之类似。

第 1088 行：激活所有特定于硬件的身份识别例程。可查阅 `arch/xxx/machine-default/setup.c`。

第 1089 行：识别特定的处理器。

第 1103~1116 行：获取系统的引导参数。

第 1118~1122 行：如果在 `arch/<arch>/defconfig` 中有相应设置，则获取 RAM 盘。

第 1124~1127 行：初始化可扩展固件接口 (Extensible Firmware Interface) (如果在 `defconfig` 中有相应设置的话)，或仅输出 BIOS 内存映像。

第 1129 行：保存引导时增强型磁盘驱动器的参数。

第 1133~1141 行：初始化由 BIOS 提供的内存映像中的内存管理结构。

第 1143 行：开始分析 Linux 命令行 (详情可查阅 `arch/<arch>/kernel/setup.c`)。

第 1145 行：初始化/保留引导内存 (详情可查阅 `arch/i386/kernel/setup.c`)。

第 1153~1155 行：获取 SMP 初始化时所需的页面，或初始化 8 M 以上的内存分页，8 M 以内的页面已经在 `head.S` 中被初始化 (详情可查阅 `arch/i386/mm/init.c`)。

第 1157~1167 行：让 `printk()` 运行，即使此时尚未完成控制台的所有初始化过程也是如此。

第 1170 行：DMI (Desktop Management Interface, 桌面管理界面)，用于从 BIOS 收集特定系统硬件的配置信息 (可查阅 `arch/i386/kernel/dmi_scan.c`)。

第 1172~1174 行：如果配置程序调用该函数，则寻找由命令行给出的 APIC (详情可查阅 `arch/i386/machine-generic/probe.c`)。

第 1175~1176 行：如果使用可扩展固件接口，则重新映射 EFI 内存映像 (参见 `arch/i386/kernel/efi.c`)。

第 1181 行：寻找局部 APIC 和 I/O APIC。(参见 `arch/i386/kernel/acpi/boot.c`) 定位并检验系统描述符表 (参见 `drivers/acpi/tables.c`)。想要更好地理解 ACPI 的话，可以在网上查阅 ACPI4LINUX 项目。

第 1183~1186 行：扫描 SMP 的配置 (参见 `arch/i386/kernel/mpparse.c`)。本节也可以将 ACPI 用做配置信息。

第 1188 行：为标准资源请求 I/O 和内存空间。(如果想了解如何注册资源，可查阅 `arch/i386/`

kernel/std_resources.c。)

第 1190~1197 行: 设置 VGA 控制台转换结构。(参见 drivers/video/console/vgacon.c)

PowerPC 中一个类似于 setup_arch(), 但更简练的函数位于文件 arch/ppc/kernel/setup.c 中。结构体 ppc_md 的大部分内容由该函数来初始化。调用 arch/ppc/platforms/pmac_feature.c 中的函数 pmac_feature_init() 进行初步检测并初始化 pmac 硬件。

8.5.5 调用 setup_per_cpu_areas()

第 409 行: 例程 setup_per_cpu_areas() 用于设置多处理器环境。如果 Linux 内核编译时不支持 SMP, 那么该函数什么都不做, 如下所示:

```
-----
init/main.c
317 static inline void setup_per_cpu_areas(void) { }
-----
```

如果 Linux 内核编译时支持 SMP, 那么该函数定义如下:

```
-----
init/main.c
327 static void __init setup_per_cpu_areas(void)
328 {
329     unsigned long size, i;
330     char *ptr;
331     /* Created by linker magic */
332     extern char __per_cpu_start[], __per_cpu_end[];
333
334     /* Copy section for each CPU (we discard the original) */
335     size = ALIGN(__per_cpu_end - __per_cpu_start, SMP_CACHE_BYTES);
336 #ifdef CONFIG_MODULES
337     if (size < PERCPU_ENOUGH_ROOM)
338         size = PERCPU_ENOUGH_ROOM;
339 #endif
340
341     ptr = alloc_bootmem(size * NR_CPUS);
342
343     for (i = 0; i < NR_CPUS; i++, ptr += size) {
344         __per_cpu_offset[i] = ptr - __per_cpu_start;
345         memcpy(ptr, __per_cpu_start, __per_cpu_end - __per_cpu_start);
346     }
347 }
-----
```

第 329~332 行: 初始化管理连续内存块的变量。在链接期间, “链接器魔数” 变量在相应体系结构的内核目录下定义 (例如, arch/i386/kernel/vmlinux.lds.s)。

第 334~341 行: 确定单个 CPU 所需内存的大小, 并为系统中每个 CPU 分配这样的内存, 就像一整块连续的内存一样。

第 343~346 行: 在新分配的内存间循环, 初始化每个 CPU 的内存块。在概念上, 我们取出了对单个 CPU 有效的一块数据 (从 __per_cpu_start 到 __per_cpu_end), 并将它复制给系统

中的每个 CPU，通过这种方式，每个 CPU 都可以操作自己的数据。

8.5.6 调用 `smp_prepare_boot_cpu()`

第 415 行：与 `smp_per_cpu_areas()` 类似，当 Linux 内核不支持 SMP 时，函数 `smp_prepare_boot_cpu()` 什么都不做：

```
-----
include/linux/smp.h
106 #define smp_prepare_boot_cpu()    do {} while (0)
-----
```

然而，如果 Linux 内核编译时支持 SMP，那么必须允许引导的 CPU 访问其控制台驱动程序，以及刚初始化过的每个 CPU 的存储器。可以通过设置 CPU 的位掩码来实现这一功能。

CPU 的位掩码定义如下：

```
-----
include/asm-generic/cpumask.h
10 #if NR_CPUS > BITS_PER_LONG && NR_CPUS != 1
11 #define CPU_ARRAY_SIZE    BITS_TO_LONGS(NR_CPUS)
12
13 struct cpumask
14 {
15     unsigned long mask[CPU_ARRAY_SIZE];
16 };
-----
```

这就意味着与平台无关的位掩码包含与系统中 CPU 个数相同的位数。

`smp_prepare_boot_cpu()` 在 Linux 内核的与体系结构相关的部分实现，读者很快就会发现，i386 系统中和 PPC 系统中这部分代码完全一样：

```
-----
arch/i386/kernel/smpboot.c
66 /* bitmap of online cpus */
67 cpumask_t cpu_online_map;
...
70 cpumask_t cpu_callout_map;
...
1341 void __devinit smp_prepare_boot_cpu(void)
1342 {
1343     cpu_set(smp_processor_id(), cpu_online_map);
1344     cpu_set(smp_processor_id(), cpu_callout_map);
1345 }
-----

arch/ppc/kernel/smp.c
49 cpumask_t cpu_online_map;
50 cpumask_t cpu_possible_map;
...
331 void __devinit smp_prepare_boot_cpu(void)
332 {
```

```

333  cpu_set(smp_processor_id(), cpu_online_map);
334  cpu_set(smp_processor_id(), cpu_possible_map);
335  }

```

在这两个函数中, `cpu_set()` 仅在类型为 `cpumask_t` 的位图中设置 `smp_processor_id()` 位。这就意味着将该位的值置为 1。

8.5.7 调用 `sched_init()`

第 422 行: 在系统运行的过程中, 调用 `sched_init()` 标志着所有对象的初始化, 调度程序通过操作这些对象来管理 CPU 时间的分配。记住, 此时仅存在一个进程, 就是 `init` 进程, 该进程正在执行 `sched_init()`:

```

-----
kernel/sched.c
3896 void __init sched_init(void)
3897 {
3898     runqueue_t *rq;
3899     int i, j, k;
3900
3901     ...
3919     for (i = 0; i < NR_CPUS; i++) {
3920         prio_array_t *array;
3921
3922         rq = cpu_rq(i);
3923         spin_lock_init(&rq->lock);
3924         rq->active = rq->arrays;
3925         rq->expired = rq->arrays + 1;
3926         rq->best_expired_prio = MAX_PRIO;
3927
3928         ...
3938         for (j = 0; j < 2; j++) {
3939             array = rq->arrays + j;
3940             for (k = 0; k < MAX_PRIO; k++) {
3941                 INIT_LIST_HEAD(array->queue + k);
3942                 __clear_bit(k, array->bitmap);
3943             }
3944             // delimiter for bitsearch
3945             __set_bit(MAX_PRIO, array->bitmap);
3946         }
3947     }
3948     /*
3949     * We have to do a little magic to get the first
3950     * thread right in SMP mode.
3951     */
3952     rq = this_rq();
3953     rq->curr = current;
3954     rq->idle = current;
3955     set_task_cpu(current, smp_processor_id());
3956     wake_up_forked_process(current);
3957
3958     /*

```

```

3959  * The boot idle thread does lazy MMU switching as well:
3960  */
3961  atomic_inc(&init_mm.mm_count);
3962  enter_lazy_tlb(&init_mm, current);
3963  }

```

第 3919~3926 行：初始化每个 CPU 的运行队列：活动队列、过期队列、以及自旋锁都在此时被初始化。回顾第 7 章，函数 `spin_lock_init()` 将自旋锁设置为 1，表示数据对象已经被解锁。

图 8-13 描述了已初始化的运行队列。

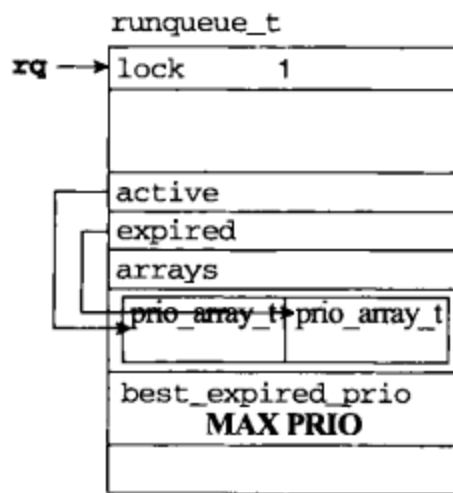


图 8-13 已初始化的运行队列 rq

第 3938~3947 行：对每个可能的优先级而言，初始化与优先级相关的链表并清除位图中所有的位，表示该队列中没有任何进程。（如果读者觉得此处不好理解的话，可以查阅图 8-14。另外，查阅第 7 章，可回顾调度程序是如何管理其运行队列的。）这段代码只是确保引入进程的一切工作都准备就绪了。从第 3947 行开始，调度程序已经就位，并知晓不存在任何进程，此时，它忽略了当前进程和空闲进程。

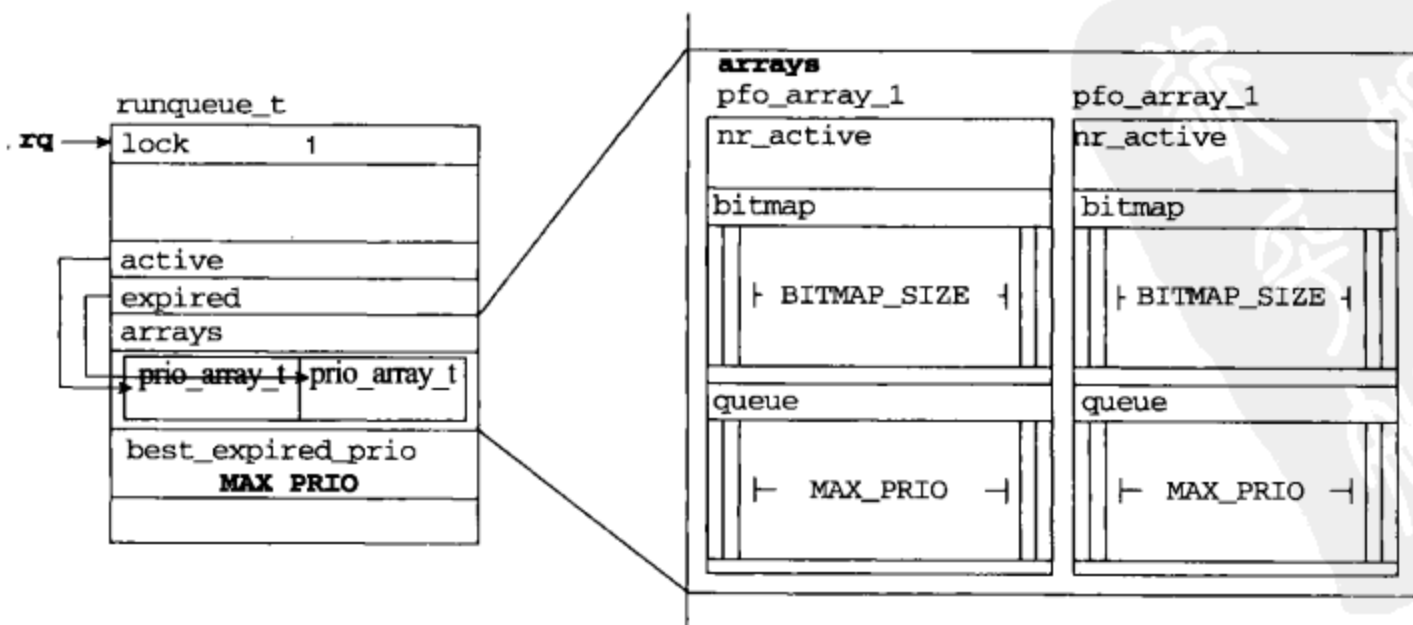


图 8-14 `rq->arrays`

第 3952~3956 行: 将当前进程加入当前 CPU 的运行队列, 并调用函数 `wake_up_forked_process()`, 以便初始化当前进程并通知调度程序。现在, 调度程序知道确实有一个进程了, 即 `init` 进程。

第 3961~3962 行: 当启用 MMU 交换技术时, 它允许多处理器 Linux 系统以更快的速度执行上下文切换。作为转换后备缓冲区, TLB 包含最近的页面转换地址。MMU 交换技术花费很长时间来刷新 TLB, 因此, 如果可能的话就交换 TLB 中的内容。函数 `enter_lazy_tlb()` 确保多 CPU 系统中没有 CPU 正在使用 `mm_struct init_mm`, 因而可以从容地进行交换。在单处理器系统中, 这是一个空函数。

上述代码忽略了用于处理 SMP 初始化的部分。概括起来, 那部分代码用于引导每个 CPU 获得默认的设置, 这些设置也是支持负载平衡、群组调度和线程迁移功能所必需的。为了简单起见, 此处忽略了这部分内容。

8.5.8 调用 `build_all_zonelists()`

第 424 行: 函数 `build_all_zonelists()` 根据页面区类型 `ZONE_DMA`、`ZONE_NORMAL` 和 `ZONE_HIGHMEM` 来划分内存空间。在第 6 章中曾提到, 页面区是物理存储器的线性划分, 主要用于突破硬件的限制, 只要知道这些页面区就是在该函数中创建的就足够了。页面区创建后, 页面就存储到属于页面区的页框中。

调用函数 `build_all_zonelists()`, 引入 `numnodes` 和 `NODE_DATA`。全局变量 `numnodes` 用于保存物理内存的节点 (或分区) 数。

分区是根据 CPU 的访问时间而定的。注意, 页表在此处就已经完全创建好了:

```
-----
mm/page_alloc.c
1345 void __init build_all_zonelists(void)
1346 {
1347     int i;
1348
1349     for(i = 0 ; i < numnodes ; i++)
1350         build_zonelists(NODE_DATA(i));
1351     printk("Built %i zonelists\n", numnodes);
1352 }
-----
```

函数 `build_all_zonelists()` 为每个结点调用一次 `build_zonelists()`, 打印出所创建的页面区列表数后, 该函数就结束了。本书并不深入探讨关于节点的更多内容。在单 CPU 的例子中, 只要知道 `numnodes` 等于 1 并且每个节点都有这三种页面区类型就足够了。宏 `NODE_DATA` 则从节点描述符列表中返回该节点的描述符。

8.5.9 调用 `page_alloc_init`

第 425 行: 函数 `page_alloc_init()` 仅向通告程序链 (notifier chain)^① 注册一个函数。已注

^① 在第 2 章中已经介绍过通告程序链。

册的函数 `page_alloc_cpu_notify()` 是一个涉及动态 CPU 配置的页摘除函数 (page-draining function) ^①。

动态 CPU 配置是指在 Linux 系统运行期间注册和撤销 CPU，涉及的事件被看做是“热插拔 CPU”。虽然从技术上讲，CPU 根本不是在机器操作期间插入和移除的，但在某些系统中可以开启和关闭，例如 IBM 的 p-Series690。让我们来看看这个函数：

```
-----
mm/page_alloc.c
1787 #ifdef CONFIG_HOTPLUG_CPU
1788 static int page_alloc_cpu_notify(struct notifier_block *self,
1789     unsigned long action, void *hcpu)
1790 {
1791     int cpu = (unsigned long)hcpu;
1792     long *count;
1793
1794     if (action == CPU_DEAD) {
1795         ...
1796         count = &per_cpu(nr_pagecache_local, cpu);
1797         atomic_add(*count, &nr_pagecache);
1798         *count = 0;
1799         local_irq_disable();
1800         __drain_pages(cpu);
1801         local_irq_enable();
1802     }
1803     return NOTIFY_OK;
1804 }
1805 #endif /* CONFIG_HOTPLUG_CPU */
1806
1807 void __init page_alloc_init(void)
1808 {
1809     hotcpu_notifier(page_alloc_cpu_notify, 0);
1810 }
-----
```

第 1809 行：本行向通告程序链 `hotcpu_notifier` 注册例程 `page_alloc_cpu_notify()`。例程 `hotcpu_notifier()` 创建一个结构体 `notifier_block`，该结构体指向函数 `page_alloc_cpu_notify()`，且其优先级为 0，然后在通告程序链 `cpu_chain`（位于 `kernel/cpu.c`）中注册一个对象。

第 1788 行：正如在第 2 章中所阐述的那样，`page_alloc_cpu_notify()` 有相应的通告程序调用参数，这个特定于系统的指针指向一个表示 CPU 序号的整数。

第 1794~1802 行：若该 CPU 已经死亡，则应该释放其所占用的页面。当 CPU 被杀死时，其行为就被设置为 `CPU_DEAD`。（可参阅同一文件中的函数 `drain_pages()`。）

8.5.10 调用 `parse_args()`

第 427 行：函数 `parse_args()` 用于解析传递给 Linux 内核的参数。

例如，`nfsroot` 是为无磁盘的系统设置 NFS 根文件系统的内核参数。读者可以在文件

^① 页摘除指的是如果 CPU 将不再被使用，那么就删除正在被该 CPU 使用的那些页。

Documentation/kernel-parameters.txt 中找到内核参数的完整列表。

```
-----
kernel/params.c
116 int parse_args(const char *name,
117     char *args,
118     struct kernel_param *params,
119     unsigned num,
120     int (*unknown)(char *param, char *val))
121 {
122     char *param, *val;
123
124     DEBUGP("Parsing ARGS: %s\n", args);
125
126     while (*args) {
127         int ret;
128
129         args = next_arg(args, &param, &val);
130         ret = parse_one(param, val, params, num, unknown);
131         switch (ret) {
132             case -ENOENT:
133                 printk(KERN_ERR "%s: Unknown parameter '%s'\n",
134                     name, param);
135                 return ret;
136             case -ENOSPC:
137                 printk(KERN_ERR
138                     "%s: '%s' too large for parameter '%s'\n",
139                     name, val ?: "", param);
140                 return ret;
141             case 0:
142                 break;
143             default:
144                 printk(KERN_ERR
145                     "%s: '%s' invalid for parameter '%s'\n",
146                     name, val ?: "", param);
147                 return ret;
148         }
149     }
150
151     /* All parsed OK. */
152     return 0;
153 }
-----
```

第 116~125 行: 传递给函数 `parse_args()` 的参数如下:

- ❑ **name**。当内核试图解析其参数变量时, 如果发生任何错误则显示一个字符串。在标准操作中, 这就意味着显示一条错误信息 “Booting Kernel: Unknown parameter X”。
- ❑ **args**。内核参数以 `foo=bar`、`bar2 baz=fuz wix` 的形式列出。
- ❑ **params**。指向内核参数的数据结构, 该结构体保存特定内核的所有有效参数。由于编译内核的方式不同, 有些参数可能存在, 有些可能不存在。
- ❑ **num**。这是该特定内核中的参数数目, 而不是 `args` 中的参数数目。

□ **unknown**。如果内核参数是未经验证而指定的，则 **unknown** 指向一个要调用的函数。

第 126~153 行：分解字符串 **args**，并将 **param** 设置为指向第一个参数的指针，**val** 为第一个参数值（如果有参数的话，否则 **val** 可以为空），这一操作是通过函数 **next_args()** 来实现的（例如，第一次用值为 **foo=bar,bar2 baz=fuz wix** 的参数来调用 **next_args()**）。我们将 **param** 设置为 **foo**，而 **val** 为 **bar,bar2**。**bar2** 后面的空间用 **\0** 来覆盖，**args** 则指向 **baz** 的起始字符。

可以看到，指针 **param** 和 **val** 被传给了 **parse_one()**，该函数用于设置实参的数据结构：

```
-----
kernel/params.c
46 static int parse_one(char *param,
47     char *val,
48     struct kernel_param *params,
49     unsigned num_params,
50     int (*handle_unknown)(char *param, char *val))
51 {
52     unsigned int i;
53
54     /* Find parameter */
55     for (i = 0; i < num_params; i++) {
56         if (parameq(param, params[i].name)) {
57             DEBUGP("They are equal! Calling %p\n",
58                 params[i].set);
59             return params[i].set(val, &params[i]);
60         }
61     }
62
63     if (handle_unknown) {
64         DEBUGP("Unknown argument: calling %p\n", handle_unknown);
65         return handle_unknown(param, val);
66     }
67
68     DEBUGP("Unknown argument '%s'\n", param);
69     return -ENOENT;
70 }
-----
```

第 46~54 行：这些参数与函数 **parse_args()** 中描述的指向部分参数的 **param** 和 **val** 相同。

第 55~61 行：分解已定义的内核参数，看看是否有和 **param** 匹配的参数。如果找到一个匹配的参数，就可以使用 **val** 来调用相关的集合函数。因此，集合函数可以处理多个或 0 个参数。

第 62~66 行：如果没有找到内核参数，就通过 **parse_args()** 来调用函数 **handle_unknown()**。

args 中每个指定的参数-值组合都调用过函数 **parse_one()** 后，内核参数就设置完毕，为继续启动 Linux 内核做好了准备。

8.5.11 调用 **trap_init()**

第 431 行：在第 3 章中已介绍过异常和中断。函数 **trap_init()** 在 x86 体系结构中是针对中

断处理操作的。简而言之,该函数初始化 x86 硬件引用的表,表中每个元素都有一个函数用于处理与内核和用户相关的问题,例如无效指令或引用了当前不在内存的页面。虽然 PowerPC 也可能存在相同的问题,但它处理这些问题的方式稍有不同(同样,所有相关内容已经在第 3 章讨论过了)。

8.5.12 调用 rcu_init()

第 432 行:函数 rcu_init()初始化 Linux 2.6 内核的读-复制-更新(Read-Copy-Update, RCU)子系统。RCU 控制访问临界区的代码,与芯片速度相比,当获取锁的代价变得非常高时,RCU 强制系统中的互斥现象。本书并不讨论 Linux 中 RCU 的实现,分析代码时偶尔会提到调用 RCU 子系统,但并不考虑其具体细节。有关 Linux 中 RCU 子系统的更多信息,可参阅 Linux Scalability Effort 的主页 <http://lse.sourceforge.net/locking/rcupdate.html>;

```
-----
kernel/rcupate.c
297 void __init rcu_init(void)
298 {
299     rcu_cpu_notify(&rcu_nb, CPU_UP_PREPARE,
300         (void *) (long) smp_processor_id());
301     /* Register notifier for non-boot CPUs */
302     register_cpu_notifier(&rcu_nb);
303 }
-----
```

8.5.13 调用 init_IRQ()

第 433 行: arch/i386/kernel/i8259.c 中的函数 init_IRQ()用于初始化硬件中断控制器和中断向量表,如果是 x86 体系结构的话,还要初始化系统定时器。第 3 章详细讨论了 x86 和 PPC 的中断,并以实时时钟为例:

```
-----
arch/i386/kernel/i8259.c
410 void __init init_IRQ(void)
411 {
412     int i;
413     ...
422     for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
423         int vector = FIRST_EXTERNAL_VECTOR + i;
424         if (i >= NR_IRQS)
425             break;
426         ...
430         if (vector != SYSCALL_VECTOR)
431             set_intr_gate(vector, interrupt[i]);
432     }
433     ...
437     intr_init_hook();
438     ...
443     setup_timer();
444     ...
-----
```

```

449 if (boot_cpu_data.hard_math && !cpu_has_fpu)
450     setup_irq(FPU_IRQ, &fpu_irq);
451 }

```

第 422~432 行：初始化中断向量。这就将 x86 的（硬件）IRQ 与相应的中断处理程序关联起来了。

第 437 行：设置特定于机器的 IRQ，例如高级可编程中断控制器（APIC）。

第 443 行：初始化定时器时钟。

第 449~450 行：需要的话，为 FPU 设置中断请求。

以下是函数 `init_IRQ()` 在 PPC 中的实现：

```

-----
arch/ppc/kernel/irq.c
700 void __init init_IRQ(void)
701 {
702     int i;
703
704     for (i = 0; i < NR_IRQS; ++i)
705         irq_affinity[i] = DEFAULT_CPU_AFFINITY;
706
707     ppc_md.init_IRQ();
708 }
-----

```

第 704 行：在多处理器系统中，中断对特定的处理器有吸引力。

第 707 行：在 PowerMac 平台上，该例程位于文件 `arch/ppc/platforms/pmac_pic.c` 中，用于设置 I/O 控制器的可编程中断控制器（PIC）部分。

8.5.14 调用 `softirq_init()`

第 436 行：函数 `softirq_init()` 为作为引导的 CPU 接受来自 tasklet 的通知做好准备。接下来看看该函数的内部结构：

```

-----
kernel/softirq.c
317 void __init softirq_init(void)
318 {
319     open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
320     open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
321 }
...
327 void __init softirq_init(void)
328 {
329     open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
330     open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
331     tasklet_cpu_notify(&tasklet_nb, (unsigned long)CPU_UP_PREPARE,
332                       (void *) (long)smp_processor_id());
333     register_cpu_notifier(&tasklet_nb);
334 }
-----

```

第 319~320 行: 当获得 TASKLET_SOFTIRQ 或 HI_SOFTIRQ 中断时, 初始化要执行的操作。当传入一个 NULL 时, 就是告诉 Linux 内核去调用函数 tasklet_action(NULL) 和 tasklet_hi_action(NULL) (分别是第 319 行和第 320 行所列的情况)。以下是函数 open_softirq() 的实现, 它说明了 Linux 内核如何存放 tasklet 的初始化信息。

```
-----
kernel/softirq.c
177 void open_softirq(int nr, void (*action)(struct softirq_action*),
void * data)
178 {
179     softirq_vec[nr].data = data;
180     softirq_vec[nr].action = action;
181 }
-----
```

8.5.15 调用 time_init()

第 437 行: 函数 time_init() 选择并初始化系统定时器。与 trap_init() 类似, 该函数非常依赖于体系结构, 第 3 章探讨定时器中断时对此进行了讨论。系统定时器为 Linux 提供了一个临时视图, 当任务应当运行时, 允许它调度该任务并确定运行多长时间。Intel 的 HPET (High Performance Event Timer, 高性能事件定时器) 将逐渐取代 8254 PIT 和 RTC。HPET 采用内存映射 I/O 的方式, 这就意味着访问 HPET 控制寄存器就像访问内存地址一样。访问 I/O 区域时必须配置恰当的存储器。如果 arch/i386/defconfig.h 中有相应设置, 就得推迟函数 time_init() 的执行, 直到函数 mem_init() 设置完内存区。请参阅以下代码:

```
-----
arch/i386/kernel/time.c
376 void __init time_init(void)
377 {
...
378 #ifdef CONFIG_HPET_TIMER
379     if (is_hpet_capable()) {
380         late_time_init = hpet_time_init;
381         return;
382     }
...
387 #endif
388     xtime.tv_sec = get_cmos_time();
389     wall_to_monotonic.tv_sec = -xtime.tv_sec;
390     xtime.tv_nsec = (INITIAL_JIFFIES % HZ) * (NSEC_PER_SEC / HZ);
391     wall_to_monotonic.tv_nsec = -xtime.tv_nsec;
392
393     cur_timer = select_timer();
394     printk(KERN_INFO "Using %s for high-res timesource\n", cur_timer->name);
395
396     time_init_hook();
397 }
-----
```

第 379~387 行：如果配置了 HPET，函数 `time_init()` 就必须在初始化内存之后才能运行。函数 `late_time_init()` 的代码（见第 358~373 行）与函数 `time_init()` 相同。

第 388~391 行：初始化 `xtime` 时间结构体，用于保存时刻。

第 393 行：选择初始化的第一个定时器。这一步可以跳过。（详情请参见 `arch/i386/kernel/timers/timer.c`。）

8.5.16 调用 `console_init()`

第 444 行：计算机控制台是内核（以及系统中其他部分）输出信息的设备，同时也有注册（`login`）功能。根据系统的不同，控制台可以位于监视器上或经由串口实现。函数 `console_init()` 是初始化控制台设备的早期函数调用，它被允许在引导时报告状态信息：

```
-----
drivers/char/tty_io.c
2347 void __init console_init(void)
2348 {
2349     initcall_t *call;
2350     ...
2352     (void) tty_register_ldisc(N_TTY, &tty_ldisc_N_TTY);
2353     ...
2358 #ifdef CONFIG_EARLY_PRINTK
2359     disable_early_printk();
2360 #endif
2361     ...
2366     call = &__con_initcall_start;
2367     while (call < &__con_initcall_end) {
2368         (*call)();
2369         call++;
2370     }
2371 }
-----
```

第 2352 行：设置线路规程。

第 2359 行：需要时保留对早期 `printk` 的支持。完全初始化系统控制台之前，支持早期 `printk` 将允许系统在引导过程中报告状态信息，至少，它还能逐一初始化串口（如 `ttyS0`）和系统的 VGA。对早期 `printk` 的支持始于函数 `setup_arch()`。（详情可参阅本节对第 408 行代码的讨论和文件 `/kernel/printk.c`，以及 `/arch/i386/kernel/early_printk.c`。）

第 2366 行：初始化控制台。

8.5.17 调用 `profile_init()`

第 447 行：函数 `profile_init()` 为内核分配内存以存储分析数据（`profiling data`）。“分析”是一个术语，在计算机科学中用于描述在程序运行期间收集的数据。分析数据常用于分析程序的性能，或研究正在运行的程序（此处是指 Linux 内核本身）：

```
-----
kernel/profile.c
30 void __init profile_init(void)
```



```

31 {
32     unsigned int size;
33
34     if (!prof_on)
35         return;
36
37     /* only text is profiled */
38     prof_len = _etext - _stext;
39     prof_len >>= prof_shift;
40
41     size = prof_len * sizeof(unsigned int) + PAGE_SIZE - 1;
42     prof_buffer = (unsigned int *) alloc_bootmem(size);
43 }

```

第 34~35 行: 如果没有开启内核分析功能, 则什么都不做。

第 38~39 行: `_etext` 和 `_stext` 在 `kernel/head.S` 中定义, 根据它们的值来确定所要存放的数据的长度, 然后通过定义为内核参数的 `prof_shift` 来改变这个数据长度。

第 41~42 行: 为存储分析数据分配连续的内存块, 其大小由内核参数来确定。

8.5.18 调用 `local_irq_enable()`

第 448 行: `local_irq_enable()` 允许中断当前 CPU, 它常与 `local_irq_disable()` 成对使用。在早先版本的内核中, 函数 `sti()` 和 `cli()` 也有相同的功能。虽然这些宏仍然需要转化为函数 `sti()` 和 `cli()`, 但此处要注意的是关键词 `local`, 这些宏仅作用于当前正在运行的处理器。

```

-----
include\asm-i386\system.h

446 #define local_irq_disable() __asm__ __volatile__ ("cli": : : "memory")
447 #define local_irq_enable() __asm__ __volatile__ ("sti": : : "memory")
-----

```

第 446~447 行: 查阅 2.4 节, 引号中的语句就是汇编指令, 且内存位于已修改列表中。

8.5.19 配置 `initrd`

第 449~456 行: 这条 `#ifdef` 语句是对 `initrd` (即原始的 RAM 盘) 进行完全的检查。

系统使用 `initrd` 来加载内核, 并挂载原始的 RAM 盘作为根文件系统。程序可以从这个 RAM 盘上运行, 当时机成熟时, 就可以挂载新的根文件系统, 例如硬盘驱动器上的某个文件系统, 并卸载原始的 RAM 盘。

该操作仅能确保指定的原始 RAM 盘有效, 否则将 `initrd_start` 设置为 0, 以通知内核不要使用原始的 RAM 盘^①。

8.5.20 调用 `mem_init()`

第 457 行: 对 x86 和 PPC 而言, `mem_init()` 的功能都是找到所有空闲页面并发送该信息到

^① 详情可查阅 `Documentation/initrd.txt`。

控制台。第4章曾提到 Linux 内核将可用内存分为多个页面区。现在, Linux 有3个页面区。

- **zone_DMA**。16 M 以内的内存。
- **zone_Normal**。从 16 M 开始但小于 896 M 的内存。(内核使用最后的 128 M)
- **zone_HIGHMEM**。大于 1 G 的内存。

函数 `mem_init()` 找到所有页面区中空闲页面的总数, 并打印出关于内存初始状态的内核信息。该函数管理早期的内存分配数据, 因此它和体系结构相关。虽然是执行同一个任务, 但每种体系结构都提供了自己的函数来实现这一功能。接下来, 首先看看 x86 是如何实现这一函数的, 之后再讨论该函数在 PPC 中的实现:

```
-----
arch/i386/mm/init
445 void __init mem_init(void)
446 {
447     extern int ppro_with_ram_bug(void);
448     int codesize, reservedpages, datasize, initsize;
449     int tmp;
450     int bad_ppro;
451     ...
459 #ifdef CONFIG_HIGHMEM
460     if (PKMAP_BASE+LAST_PKMAP*PAGE_SIZE >= FIXADDR_START) {
461         printk(KERN_ERR "fixmap and kmap areas overlap - this will crash\n");
462         printk(KERN_ERR "pkstart: %lxh pkend: %lxh fixstart %lxh\n",
463             PKMAP_BASE, PKMAP_BASE+LAST_PKMAP*PAGE_SIZE, FIXADDR_START);
464         BUG();
465     }
466 #endif
467
468     set_max_mapnr_init();
469     ...
476 /* this will put all low memory onto the freelists */
477     totalram_pages += __free_all_bootmem();
478
479
480     reservedpages = 0;
481     for (tmp = 0; tmp < max_low_pfn; tmp++)
482     ...
485     if (page_is_ram(tmp) && PageReserved(pfn_to_page(tmp)))
486         reservedpages++;
487
488     set_highmem_pages_init(bad_ppro);
489     codesize = (unsigned long) &_amp;etext - (unsigned long) &_amp;text;
490     datasize = (unsigned long) &_amp;edata - (unsigned long) &_amp;etext;
491     initsize = (unsigned long) &_amp;__init_end - (unsigned long) &_amp;__init_begin;
492
493
494     kclist_add(&kcore_mem, __va(0), max_low_pfn << PAGE_SHIFT);
495     kclist_add(&kcore_vmalloc, (void *)VMALLOC_START,
496         VMALLOC_END-VMALLOC_START);
497
498     printk(KERN_INFO "Memory: %luk/%luk available (%dk kernel code, %dk
reserved, %dk data, %dk init, %ldk highmem)\n",
499         (unsigned long) nr_free_pages() << (PAGE_SHIFT-10),
```

```

500     num_physpages << (PAGE_SHIFT-10),
501     codesize >> 10,
502     reservedpages << (PAGE_SHIFT-10),
503     datasize >> 10,
504     initsize >> 10,
505     (unsigned long) (totalhigh_pages << (PAGE_SHIFT-10))
506     );
...
521 #ifndef CONFIG_SMP
522     zap_low_mappings();
523 #endif
524 }

```

第 459 行: 本行实现的是简单的错误检查功能, 从而使固定映像不会和内核映像重叠。

第 469 行: 函数 `set_max_mapnr_init()` (详情请参阅 `arch/i386/mm/init.c`) 仅设置 `num_physpages` 的值, 这是一个用于保存可用页面数的全局变量 (在 `mm/memory.c` 中定义)。

第 477 行: `__free_all_bootmem()` 用于标记所有正在释放的低端内存页面。引导时, 所有页面都被保留。此处, 将在引导阶段的稍后时刻释放可用的低端内存页面。该函数的调用流程参见图 8-15。

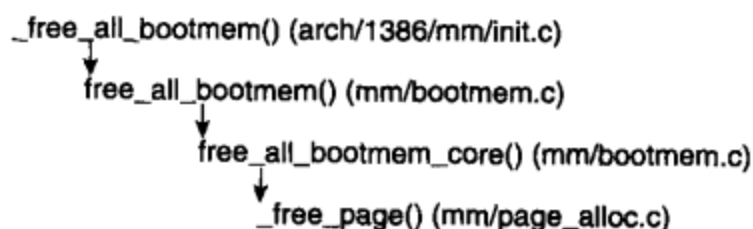


图 8-15 `__free_all_bootmem()` 调用的层次

让我们来看看 `free_all_bootmem_core()` 的核心部分, 以便理解它究竟是如何工作的:

```

-----
mm/bootmem.c
257 static unsigned long __init free_all_bootmem_core(pg_data_t *pgdat)
258 {
259     struct page *page;
260     bootmem_data_t *bdata = pgdat->bdata;
261     unsigned long i, count, total = 0;
...
295     page = virt_to_page(bdata->node_bootmem_map);
296     count = 0;
297     for (i = 0; i < ((bdata->node_low_pfn - (bdata->node_boot_start >>
PAGE_SHIFT)) / 8 + PAGE_SIZE - 1) / PAGE_SIZE; i++, page++) {
298         count++;
299         ClearPageReserved(page);
300         set_page_count(page, 1);
301         __free_page(page);
302     }
303     total += count;
304     bdata->node_bootmem_map = NULL;
305
306     return total;

```

```
307 }
```

对所有可用的低端内存页面而言,需要清除页结构中 flags 字段的 PG_reserved 标志^①。接下来,将页结构的 count 字段设置为 1,表示该页正在使用,然后调用 __free_page(),以便将该页传给伙伴分配程序。若你已回想起第 4 章阐述过的伙伴系统,就可以解释该函数如何释放一个页面并将其加入到空闲链表了。

函数 __free_all_bootmem() 返回低端内存的可用页面数,该值被加入 totalram_pages (在 mm/page_alloc.c 中定义的无符号长整型数) 的运行计数中。

第 480~486 行: 这些行更新预留页面的计数。

第 488 行: 调用 set_highmem_pages_init(), 标记高端内存页面的初始化。图 8-16 解释了该函数的调用层次。

```
set_highmem_pages_init()(arch/i386/mm/init.c)
    ↓
one_highpage_init()(arch/i386/mm/init.c)
```

图 8-16 highmem_pages_init 的调用层次

让我们来看看 one_highpage_init() 中执行的大部分代码:

```
arch/i386/mm/init.c
253 void __init one_highpage_init(struct page *page, int pfn, int bad_ppro)
254 {
255     if (page_is_ram(pfn) && !(bad_ppro && page_kills_ppro(pfn))) {
256         ClearPageReserved(page);
257         set_bit(PG_highmem, &page->flags);
258         set_page_count(page, 1);
259         __free_page(page);
260         totalhigh_pages++;
261     } else
262         SetPageReserved(page);
263 }
```

与 __free_all_bootmem() 非常相似,所有高端内存页面都有自己的由 PG_reserved 标志清除的页结构 flags 字段,也有 PG_highmem 集合,并将其 count 字段设置为 1。同时也可以调用 __free_page(), 将这些页面加入到空闲链表中,并将 totalhigh_pages 计数器加 1。

第 490~506 行: 该代码块收集并打印与存储区大小相关的信息以及可用页面数。

第 521~523 行: 函数 zap_low_mappings 刷新低端内存中初始化的 TLB 和 PGD。

函数 mem_init() 标记内存分配引导阶段的末尾,以及将贯穿整个系统的内存分配的开始。

PPC 中 mem_init() 的代码用于找到并初始化所有页面区的每个页面:

```
arch/ppc/mm/init.c
393 void __init mem_init(void)
```

① 回顾第 6 章,曾讨论过对已经在内存的页设置该标志。在早期的引导过程中,对低端内存也设置该标志。

```

394 {
395     unsigned long addr;
396     int codepages = 0;
397     int datapages = 0;
398     int initpages = 0;
399     #ifdef CONFIG_HIGHMEM
400     unsigned long highmem_mapnr;

402     highmem_mapnr = total_lowmem >> PAGE_SHIFT;
403     highmem_start_page = mem_map + highmem_mapnr;
404 #endif /* CONFIG_HIGHMEM */
405     max_mapnr = total_memory >> PAGE_SHIFT;

407     high_memory = (void *) __va(PPC_MEMSTART + total_lowmem);
408     num_physpages = max_mapnr; /* RAM is assumed contiguous */

410     totalram_pages += free_all_bootmem();

412 #ifdef CONFIG_BLK_DEV_INITRD
413     /* if we are booted from BootX with an initial ramdisk,
414        make sure the ramdisk pages aren't reserved. */
415     if (initrd_start) {
416         for (addr = initrd_start; addr < initrd_end; addr += PAGE_SIZE)
417             ClearPageReserved(virt_to_page(addr));
418     }
419 #endif /* CONFIG_BLK_DEV_INITRD */

421 #ifdef CONFIG_PPC_OF
422     /* mark the RTAS pages as reserved */
423     if (rtas_data)
424         for (addr = (ulong)__va(rtas_data);
425              addr < PAGE_ALIGN((ulong)__va(rtas_data)+rtas_size) ;
426              addr += PAGE_SIZE)
427             SetPageReserved(virt_to_page(addr));
428 #endif
429 #ifdef CONFIG_PPC_PMAC
430     if (agp_special_page)
431         SetPageReserved(virt_to_page(agp_special_page));
432 #endif
433     if (sysmap)
434         for (addr = (unsigned long)sysmap;
435              addr < PAGE_ALIGN((unsigned long)sysmap+sysmap_size) ;
436              addr += PAGE_SIZE)
437             SetPageReserved(virt_to_page(addr));

439     for (addr = PAGE_OFFSET; addr < (unsigned long)high_memory;
440          addr += PAGE_SIZE) {
441         if (!PageReserved(virt_to_page(addr)))
442             continue;
443         if (addr < (ulong) etext)

```

```
444     codepages++;
445     else if (addr >= (unsigned long)&__init_begin
446             && addr < (unsigned long)&__init_end)
447         initpages++;
448     else if (addr < (ulong) klimit)
449         datapages++;
450 }

452 #ifdef CONFIG_HIGHMEM
453 {
454     unsigned long pfn;

456     for (pfn = highmem_mapnr; pfn < max_mapnr; ++pfn) {
457         struct page *page = mem_map + pfn;

459         ClearPageReserved(page);
460         set_bit(PG_highmem, &page->flags);
461         set_page_count(page, 1);
462         __free_page(page);
463         totalhigh_pages++;
464     }
465     totalram_pages += totalhigh_pages;
466 }
467 #endif /* CONFIG_HIGHMEM */

469 printk("Memory: %luk available (%dk kernel code, %dk data, %dk init, %ldk
highmem)\n",
470        (unsigned long)nr_free_pages() << (PAGE_SHIFT-10),
471        codepages << (PAGE_SHIFT-10), datapages << (PAGE_SHIFT-10),
472        initpages << (PAGE_SHIFT-10),
473        (unsigned long) (totalhigh_pages << (PAGE_SHIFT-10)));
474 if (sysmap)
475     printk("System.map loaded at 0x%08x for debugger, size: %ld bytes\n",
476            (unsigned int)sysmap, sysmap_size);
477 #ifdef CONFIG_PPC_PMAC
478     if (agp_special_page)
479         printk(KERN_INFO "AGP special page: 0x%08lx\n", agp_special_page);
480 #endif

482 /* Make sure all our pagetable pages have page->mapping
483    and page->index set correctly. */
484 for (addr = KERNELBASE; addr != 0; addr += PGDIR_SIZE) {
485     struct page *pg;
486     pmd_t *pmd = pmd_offset(pgd_offset_k(addr), addr);
487     if (pmd_present(*pmd)) {
488         pg = pmd_page(*pmd);
489         pg->mapping = (void *) &init_mm;
490         pg->index = addr;
491     }
492 }

493 mem_init_done = 1;
494 }
```

第 399~410 行: 这些行找出可用内存的总数。如果使用了 HIGHMEM, 那么高端内存页面也要计入总数, 可以通过修改全局变量 `totalram_pages` 来反映这一点。

第 412~419 行: 如果使用了原始的 RAM 盘, 就清除引导 RAM 盘使用过的任何页面。

第 421~432 行: 如果需要的话, 可根据引导环境的不同, 为实时抽象服务和 AGP (视频) 预留所需的页面。

第 433~450 行: 需要时为系统映像预留页面。

第 452~467 行: 如果正在使用 HIGHMEM, 那么清除任何预留的页面, 并修改全局变量 `totalram_pages`。

第 469~480 行: 将内存信息输出到控制台。

第 482~492 行: 重新回到页目录并初始化每个 `mm_struct` 结构和索引。

8.5.21 调用 `late_time_init()`

第 459~460 行: 函数 `late_time_init()` 使用 HPET (详情可参见 8.5.15 节)。该函数仅用于 Intel 体系结构和 HPET, 它本质上与 `time_init()` 相同, 只不过是在内存初始化后才调用, 以允许将 HPET 映射到物理内存。

8.5.22 调用 `calibrate_delay()`

第 461 行: `init/main.c` 中的函数 `calibrate_delay()` 用于计算并打印非常著名的“BogoMips”的值, 该值用于度量处理器在一个时钟节拍内可以反复执行多少个 `delay()`。对不同速度的处理器而言, `calibrate_delay()` 允许的延迟大致相同。处理器最多能运行多快, 其结果存储在 `loop_pre_jiffy` 中, 函数 `udelay()` 和 `mdelay()` 使用该值来设置反复执行 `delay()` 的次数:

```
-----
init/main.c
void __init calibrate_delay(void)
{
    unsigned long ticks, loopbit;
    int lps_precision = LPS_PREC;

186    loops_per_jiffy = (1<<12);

    printk("Calibrating delay loop... ");
189    while (loops_per_jiffy <= 1) {
        /* wait for "start of" clock tick */
        ticks = jiffies;
        while (ticks == jiffies)
            /* nothing */;
        /* Go .. */
        ticks = jiffies;
        __delay(loops_per_jiffy);
        ticks = jiffies - ticks;
        if (ticks)
            break;
200    }
```



```

/* Do a binary approximation to get loops_per_jiffy set to equal one clock
(up to lps_precision bits) */
204 loops_per_jiffy >>= 1;
    loopbit = loops_per_jiffy;
206 while ( lps_precision-- && (loopbit >>= 1) ) {
    loops_per_jiffy |= loopbit;
    ticks = jiffies;
    while (ticks == jiffies);
    ticks = jiffies;
    __delay(loops_per_jiffy);
    if (jiffies != ticks) /* longer than 1 tick */
        loops_per_jiffy &= ~loopbit;
214 }

/* Round the value and print it */
217 printk("%lu.%02lu BogoMIPS\n",
    loops_per_jiffy/(500000/HZ),
219 (loops_per_jiffy/(5000/HZ)) % 100);
}
-----

```

第 186 行：从地址 0x800 处开始。

第 189~200 行：将 loops_per_jiffy 加倍，直到它执行函数 delay(loops_per_jiffy) 所花费的总时间超过一个 jiffy。

第 204 行：loops_per_jiffy 除以 2。

第 206~214 行：将 loops_per_jiffy 连续加上 2 的递减次幂，直至节拍数等于 jiffy。

第 217~219 行：以浮点数的形式打印出该值。

8.5.23 调用 pgtable_cache_init()

第 463 行：x86 代码块中关键的函数是系统函数 kmem_cache_create()，该函数创建一个指定的缓冲区。它的第一个参数是一个字符串，常用于在 /proc/slabinfo 中识别该函数：

```

-----
arch/i386/mm/init.c
529 kmem_cache_t *pgd_cache;
530 kmem_cache_t *pmd_cache;
531
532 void __init pgtable_cache_init(void)
533 {
534     if (PTRS_PER_PMD > 1) {
535         pmd_cache = kmem_cache_create("pmd",
536             PTRS_PER_PMD*sizeof(pmd_t),
537             0, 538             SLAB_HWCACHE_ALIGN | SLAB_MUST_H  WCACHE_ALIGN,
539             pmd_ctor,
540             NULL);
541         if (!pmd_cache)
542             panic("pgtable_cache_init(): cannot create pmd c  ache");
543     }
544     pgd_cache = kmem_cache_create("pgd",
545         PTRS_PER_PGD*sizeof(pgd_t),

```

```

546         0,
547         SLAB_HWCACHE_ALIGN | SLAB_MUST_HWCACHE_ALIGN,
548         pgd_ctor,
549         PTRS_PER_PMD == 1 ? pgd_dtor : NULL);
550     if (!pgd_cache)
551         panic("pgtable_cache_init(): Cannot create pgd cache");
552 }
-----
arch/ppc64/mm/init.c
976 void pgtable_cache_init(void)
977 {
978     zero_cache = kmem_cache_create("zero",
979     PAGE_SIZE,
980     0,
981     SLAB_HWCACHE_ALIGN | SLAB_MUST_HWCACHE_ALIGN,
982     zero_ctor,
983     NULL);
984     if (!zero_cache)
985         panic("pgtable_cache_init(): could not create zero_cache !\n");
986 }
-----

```

第 532~542 行: 创建高速缓存 pmd。

第 544~551 行: 创建高速缓存 pgd。

PPC 中有硬件辅助散列法, 因而 pgtable_cache_init() 是一个空操作函数:

```

-----
include\asmppc\pgtable.h
685 #define pgtable_cache_init() do { } while (0)
-----

```

8.5.24 调用 buffer_init()

第 472 行: fs/buffer.c 中的函数 buffer_init() 用于保存从文件系统设备获得的数据:

```

-----
fs/buffer.c
3031 void __init buffer_init(void)
3032 {
3033     int i;
3034     int nrpages;

3036     bh_cachep = kmem_cache_create("buffer_head",
3037     sizeof(struct buffer_head), 0,
3038     0, init_buffer_head, NULL);
3039     for (i = 0; i < ARRAY_SIZE(bh_wait_queue_heads); i++)
3040         init_waitqueue_head(&bh_wait_queue_heads[i].wqh);

3044     nrpages = (nr_free_buffer_pages() * 10) / 100;
3045     max_buffer_heads = nrpages * (PAGE_SIZE / sizeof(struct buffer_head));
3046     hotcpu_notifier(buffer_cpu_notify, 0);
3048 }
-----

```

第 3036 行：分配缓冲区散列表。

第 3039 行：创建散列等待队列缓冲区表。

第 3044 行：将低端内存限制到 10%。

8.5.25 调用 `security_scaffolding_startup()`

第 474 行：Linux 2.6 内核包含加载内核模块的代码，这些内核模块可实现各种安全特性。`security_scaffolding_startup()` 仅检验一个安全操作对象是否存在，如果存在则调用安全模块的初始化函数。

本书并不讨论如何创建安全模块以及作者可能要面对什么样的问题，详情可查阅 Linux 安全模块 (<http://lsm.immunix.org/>) 和 Linux 安全模块邮件列表 (<http://mail.wirex.com/mailman/listinfo/linux-security-module>)。

8.5.26 调用 `vfs_caches_init()`

第 475 行：VFS 子系统依赖于内存缓存，也叫做 SLAB 缓存，来保存它所管理的数据结构。第 4 章已经详细讨论过 SLAB 缓存。函数 `vfs_caches_init()` 初始化子系统所使用的 SLAB 缓存，图 8-17 简单展示了它所调用的主要函数的层次结构。我们将详细讨论这些函数。具体分析每个函数时，参考该层次关系将有助于理解这些函数。

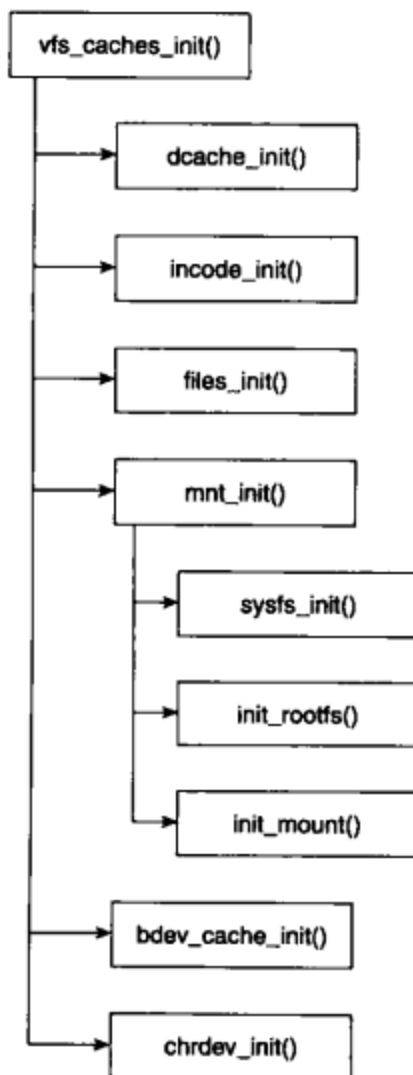


图 8-17 `vfs_caches_init()` 的调用层次

表 8-4 总结了由 `vfs_caches_init()` 和它所调用的某个函数引入的对象。

表8-4 `vfs_caches_init()`引入的对象

对 象 名	描 述
<code>names_cache</code>	全局变量
<code>filp_cache</code>	全局变量
<code>inode_cache</code>	全局变量
<code>dentry_cache</code>	全局变量
<code>mnt_cache</code>	全局变量
<code>namespace</code>	指向结构体的指针
<code>mount_hashtable</code>	全局变量
<code>root_fs_type</code>	全局变量
<code>file_system_type</code>	结构体 (参见第6章)
<code>bdev_cache</code>	全局变量

```

-----
fs/dcache.c
1623 void __init vfs_caches_init(unsigned long mempages)
1624 {
1625     names_cache = kmem_cache_create("names_cache",
1626     PATH_MAX, 0,
1627     SLAB_HWCACHE_ALIGN, NULL, NULL);
1628     if (!names_cache)
1629         panic("Cannot create names SLAB cache");
1630
1631     filp_cache = kmem_cache_create("filp",
1632     sizeof(struct file), 0,
1633     SLAB_HWCACHE_ALIGN, filp_ctor, filp_dtor);
1634     if (!filp_cache)
1635         panic("Cannot create filp SLAB cache");
1636
1637     dcache_init(mempages);
1638     inode_init(mempages);
1639     files_init(mempages);
1640     mnt_init(mempages);
1641     bdev_cache_init();
1642     chrdev_init();
1643 }
-----

```

第 1623 行: 该例程用全局变量 `num_physpages` (在 `mem_init()` 中计算该变量的值) 作为参数, 来保存在系统内存中可用的物理页面数。稍后我们将看到, 该参数值会影响 SLAB 缓存的创建。

第 1625~1629 行: 下一步就是创建内存区 `names_cache`。第 4 章详细描述了函数 `kmem_cache_create()`。该内存区保存大小为 `PATH_MAX` 的对象, `PATH_MAX` 则限定了路径名最大允许包含的字符数 (其数值在文件 `linux/limits.h` 中设置为 4096)。此处, 已创建的缓存中还没有对象, 或没有大小为 `PATH_MAX` 的内存区, 实际的内存区将在随后首次调用 `getname()` 时被分配。

第 6 章已经讨论过, 某些与文件相关的系统调用 (如 `sys_open()`) 一开始就调用了 `getname()`

例程，以便从进程的地址空间读取文件路径名，`putname()` 例程则从缓存释放对象。

如果无法创建缓存 `names_cache`，内核就会跳转到恐慌（panic）例程，从而退出该函数的控制流程。

第 1631~1635 行：下一步就是创建缓存 `filp_cache` 以及大小等于文件结构的对象。例程 `get_empty_filp()`（参见文件 `fs/file_table.c`）用于分配这个保存文件结构的对象，该例程将在创建管道或打开文件时被调用。调用例程 `file_free()`（参见文件 `fs/file_table.c`）则可回收该文件描述符对象。

第 1637 行：例程 `dcache_init()`（参见文件 `fs/dcache.c`）用于创建 SLAB 缓存，该缓存保存目录项描述符^①。缓存本身被称作 `dentry_cache`。当进程访问文件或目录时所涉及的路径名由多个路径分量组成，目录项描述符就是针对每个分量而创建的。目录项结构把文件或目录分量与其索引节点结合起来，因而通过该目录项可以更快地找到与其对应的索引节点。

第 1638 行：例程 `inode_init()`（参见文件 `fs/inode.c`）初始化索引节点散列表和由等待队列的队首节点构成的数组，该数组用于存储内核将要锁存的散列的索引节点。散列的索引节点的等待队列队首节点（`wait_queue_head_t`）存储在数组 `i_wait_queue_heads` 中，该数组将在系统的启动过程中被初始化。

此处创建 `inode_hashtable`，该表可以加速索引节点的查找。最后，SLAB 缓冲常用于保存创建的索引节点对象，称为 `inode_cache`。可以调用 `alloc_inode`（参见文件 `fs/inode.c`）来分配该缓存所在的内存区，而调用 `destroy_inode()`（参见文件 `fs/inode.c`）来释放这些内存区。

```
-----
fs/file_table.c
292 void __init files_init(unsigned long mempages)
293 {
294     int n;
...
299     n = (mempages * (PAGE_SIZE / 1024)) / 10;
300     files_stat.max_files = n;
301     if (files_stat.max_files < NR_FILE)
302         files_stat.max_files = NR_FILE;
303 }
```

第 299 行：用文件（包括其索引节点和缓存）大概占用的空间总量除以页面大小（这种情况下是 1 K），得到的值再乘以页数就得到了文件所占用的总块数。除以 10 表示其默认限制文件使用的内存不得超过可用内存的 10%。

第 301~302 行：将 `NR_FILE`（参见文件 `include/linux/fs.h`）设置为 8 192。

第 1640 行：下一个例程 `mnt_init()` 创建了用于保存 `vfsmount` 对象且名为 `mnt_cache` 的缓存，VFS 利用这些对象来挂载文件系统。该例程也创建 `mount_hashtable` 数组，该数组用于存放 `mnt_cache` 中对象引用，以便快速访问对象。然后，该例程发出调用来初始化 `sysfs` 文件系统并挂载 `root` 文件系统。接下来将考虑如何创建散列表：

① 回想一下，`dentry` 就是目录项 `directory entry` 的缩写。

```

-----
fs/namespace.c
1137 void __init mnt_init(unsigned long mempages)
{
1139     struct list_head *d;
1140     unsigned long order;
1141     unsigned int nr_hash;
1142     int i;
    ...
1149     order = 0;
1150     mount_hashtable = (struct list_head *)
1151         __get_free_pages(GFP_ATOMIC, order);
1152
1153     if (!mount_hashtable)
1154         panic("Failed to allocate mount hash table\n");
    ...
1161     nr_hash = (1UL << order) * PAGE_SIZE / sizeof(struct list_head);
1162     hash_bits = 0;
1163     do {
1164         hash_bits++;
1165     } while ((nr_hash >> hash_bits) != 0);
1166     hash_bits--;
    ...
1172     nr_hash = 1UL << hash_bits;
1173     hash_mask = nr_hash - 1;
1174
1175     printk("Mount-cache hash table entries: %d (order: %ld, %ld bytes)\n",
        nr_hash, order, (PAGE_SIZE << order));
    ...
1179     d = mount_hashtable;
1180     i = nr_hash;
1181     do {
1182         INIT_LIST_HEAD(d);
1183         d++;
1184         i--;
1185     } while (i);
    ..
1189 }
-----

```

第 1139~1144 行: 散列表队列由整个内存页面组成。在第 4 章中已经详细阐释了例程 `__get_free_pages()` 是如何工作的。简单地说, 该例程将返回指向大小为 2 的幂次个页面的内存区指针。因此, 我们分配一个页面来保存散列表。

第 1161~1173 行: 下一步就是确定表中的项数。`nr_hash` 保存散列表中链表头的次幂数 (2 的幂次)。`hash_bits` 是 `nr_hash` 中用于表示 2 的最高次幂所需的二进制位数。然后, 第 1172 行重新定义 `nr_hash` 而构成最左边的一位, 这样, 位掩码就可以根据新的 `nr_hash` 值来计算。

第 1179~1185 行: 最后, 调用宏 `INIT_LIST_HEAD` 来初始化散列表。该宏的参数是一个指针, 指向将要被初始化的新链表表头的内存区。这一过程将重复 `nr_hash` 次 (或该表可以容纳的项数)。

我们来看一个例子: 假定 `PAGE_SIZE` 是 4 KB, 而结构体 `list_head` 是 8 字节。因为 `order`

=0, nr_hash 的值变为 500; 也就是说, 在一个 4 KB 的表中可以填充 500 个 list_head 结构体。(1UL<<order) 变为已分配的页数。例如, 若 order 为 1 (表示已经要求为散列表分配 2 个页面), 0000 0001 左移一位变为 0000 0010 (在十进制计数法中是乘以 2)。接下来, 计算散列关键值所需的位数。预排该循环中的每次迭代, 可得到如下结果。

初始值: hash_bit=0, nr_hash=500。

- 第 1 次循环: hash_bit=1, (500>>1) !=0
(0001 1111 0100 >>1) = 0000 1111 1010
- 第 2 次循环: hash_bit=2, (500>>2) !=0
(0001 1111 1010 >>2) = 0000 0111 1110
- 第 3 次循环: hash_bit=3, (500>>3) !=0
(0001 1111 1010 >>3) = 0000 0011 1111
- 第 4 次循环: hash_bit=4, (500>>4) !=0
(0001 1111 1010 >>4) = 0000 0001 1111
- 第 5 次循环: hash_bit=5, (500>>5) !=0
(0001 1111 1010 >>5) = 0000 0000 1111
- 第 6 次循环: hash_bit=6, (500>>6) !=0
(0001 1111 1010 >>6) = 0000 0000 0111
- 第 7 次循环: hash_bit=7, (500>>7) !=0
(0001 1111 1010 >>7) = 0000 0000 0011
- 第 8 次循环: hash_bit=8, (500>>8) !=0
(0001 1111 1010 >>8) = 0000 0000 0001
- 第 9 次循环: hash_bit=9, (500>>9) !=0
(0001 1111 1010 >>9) = 0000 0000 0000

跳出 while 循环后, hash_bits 将减至 8, nr_hash 设为 0001 0000 0000, 而 hash_mask 被设为 0000 1111 1111。

例程 mnt_init() 在初始化 mount_hashtable 并创建 mnt_cache 后, 将发出如下 3 个调用:

```
-----
fs/namespace.c
...
1189 sysfs_init();
1190 init_rootfs();
1191 init_mount_tree();
1192 }
```

sysfs_init() 负责创建 sysfs 文件系统。init_rootfs() 和 init_mount_tree() 负责挂载根文件系统。下面将依次考虑每个例程的具体实现。

```
-----
init_rootfs()
fs/ramfs/inode.c
218 static struct file_system_type rootfs_fs_type = {
```



```

219     .name     = "rootfs",
220     .get_sb    = rootfs_get_sb,
221     .kill_sb   = kill_litter_super,
222 };
...
237 int __init init_rootfs(void)
238 {
239     return register_filesystem(&rootfs_fs_type);
240 }

```

rootfs 文件系统是内核最初挂载的文件系统，它是一个简单的空目录，稍后在内核引导的过程中挂载了真正的文件系统后，它就是多余的了。

第 218~222 行：这一代码块声明结构体 `rootfs_fs_type file_system_type`，此处只定义了这两种获得和摘除相关超级块的方法。

第 237~240 行：`init_rootfs()` 例程仅向内核注册这个 `rootfs`，这使得在内核中可以使用与文件系统类型相关的所有信息（这些信息被存储在结构体 `file_system_type` 中）。

```

-----
init_mount_tree()
fs/namespace.c
1107 static void __init init_mount_tree(void)
1108 {
1109     struct vfsmount *mnt;
1110     struct namespace *namespace;
1111     struct task_struct *g, *p;
1112
1113     mnt = do_kern_mount("rootfs", 0, "rootfs", NULL);
1114     if (IS_ERR(mnt))
1115         panic("Can't create rootfs");
1116     namespace = kmalloc(sizeof(*namespace), GFP_KERNEL);
1117     if (!namespace)
1118         panic("Can't allocate initial namespace");
1119     atomic_set(&namespace->count, 1);
1120     INIT_LIST_HEAD(&namespace->list);
1121     init_rwsem(&namespace->sem);
1122     list_add(&mnt->mnt_list, &namespace->list);
1123     namespace->root = mnt;
1124
1125     init_task.namespace = namespace;
1126     read_lock(&tasklist_lock);
1127     do_each_thread(g, p) {
1128         get_namespace(namespace);
1129         p->namespace = namespace;
1130     } while_each_thread(g, p);
1131     read_unlock(&tasklist_lock);
1132
1133     set_fs_pwd(current->fs, namespace->root,
1134                namespace->root->mnt_root);
1134     set_fs_root(current->fs, namespace->root,
1135                namespace->root->mnt_root);
1135 }
-----

```

第 1116~1123 行：初始化进程的命名空间。该结构保存指向与挂载树相关的结构及其相应目录项的指针。并分配 namespace 对象，将其计数值设置为 1，初始化 list_head 类型的 list 字段，初始化对命名空间（和挂载树）加锁的信号，将 vfsmount 结构对应的 root 字段设置为指向新分配的 vfsmount。

第 1125 行：将当前任务（即 init 任务）的进程描述符命名空间设置为指向刚分配并初始化的命名空间对象。（当前进程是进程 0。）

第 1134~1135 行：下列两个例程设置与该进程相关的 fs_struct 的四个字段的值。fs_struct 保存由这些例程设置的根目录和当前工作目录字段。

最后，分析 mnt_init 函数中究竟实现了哪些操作。现在，让我们继续来看看 vfs_mnt_init。

```
-----
1641 bdev_cache_init()
fs/block_dev.c
290 void __init bdev_cache_init(void)
291 {
292     int err;
293     bdev_cachep = kmem_cache_create("bdev_cache",
294         sizeof(struct bdev_inode),
295         0,
296         SLAB_HWCACHE_ALIGN|SLAB_RECLAIM_ACCOUNT,
297         init_once,
298         NULL);
299     if (!bdev_cachep)
300         panic("Cannot create bdev_cache SLAB cache");
301     err = register_filesystem(&bd_type);
302     if (err)
303         panic("Cannot register bdev pseudo-fs");
304     bd_mnt = kern_mount(&bd_type);
305     err = PTR_ERR(bd_mnt);
306     if (IS_ERR(bd_mnt))
307         panic("Cannot create bdev pseudo-fs");
308     blockdev_superblock = bd_mnt->mnt_sb; /* For writeback */
309 }
```

第 293~298 行：创建 SLAB 缓存 bdev_cache，该缓存用于保存 bdev_inodes。

第 301 行：注册特殊文件系统 bdev。该文件系统定义如下：

```
-----
fs/block_dev.c
294 static struct file_system_type bd_type = {
295     .name    = "bdev",
296     .get_sb  = bd_get_sb,
297     .kill_sb = kill_anon_super,
298 };
-----
```

如上所示，特殊文件系统 bdev 的 file_system_type 结构仅定义了两个例程：一个用于获取文件系统的超级块，另一个用于删除/释放超级块。读者可能会好奇为什么块设备要像文件系统一样向内核注册。在第 6 章中，我们已经看到，从技术上讲，那些并不是文件系统的系统也可

以使用文件系统的内核结构,也就是说,它们并没有挂载点,但可以使用所支持的文件系统的 VFS 内核结构。块设备就是这样一个使用 VFS 文件系统内核结构的伪文件系统。有了 bdev 后,这些特殊的文件系统通常仅需定义有限的字段,对某些特殊应用而言,并不是所有的字段都有意义。

第 304~308 行:调用 `kern_mount()` 来设置所有和挂载相关的 VFS 结构,并返回 `vfsmount` 结构。(有关如何设置全局变量 `bd_mnt` 使之指向 `vfsmount` 结构,以及设置 `blockdev_superblock`,使之指向 `vfsmount` 超级块,详情可参阅第 6 章)。

该函数用于初始化围绕驱动程序模型的字符设备对象:

```
-----
1642 chrdev_init
fs/char_dev.c
void __init chrdev_init(void)
{
433     subsystem_init(&cdev_subsys);
434     cdev_map = kobj_map_init(base_probe, &cdev_subsys);
435 }
```

8.5.27 调用 `radix_tree_init()`

第 476 行:Linux 2.6 内核使用基树 (radix tree) 来管理页高速缓存中的页面。此处,仅初始化内核空间毗邻的部分,以便存放页高速缓存的基树:

```
-----
lib/radix-tree.c
798 void __init radix_tree_init(void)
799 {
800     radix_tree_node_cachep = kmem_cache_create("radix_tree_node",
801         sizeof(struct radix_tree_node), 0,
802         SLAB_PANIC, radix_tree_node_ctor, NULL);
803     radix_tree_init_maxindex();
804     hotcpu_notifier(radix_tree_callback, 0);
-----

lib/radix-tree.c
768 static __init void radix_tree_init_maxindex(void)
769 {
770     unsigned int i;
771
772     for (i = 0; i < ARRAY_SIZE(height_to_maxindex); i++)
773         height_to_maxindex[i] = __maxindex(i);
774 }
```

请注意看看 `radix_tree_init()` 究竟是如何分配页高速缓存空间的,而 `radix_tree_init_maxindex()` 又是如何配置基树的数据存储 `height_to_maxindex[]` 的。

`hotcpu_notifier()` (见第 804 行)是指 Linux 2.6 热交换 CPU 的性能。当 CPU 被热交换时,内核就调用 `radix_tree_callback()`,该函数用于尝试释放连接到热交换 CPU 的页高速缓存。

8.5.28 调用 `signal_init()`

第 477 行: `kernel/signal.c` 中的函数 `signals_init()` 将初始化内核信号队列。

```
-----
fs/buffer.c
2565 void __init signals_init(void)
2566 {
2567     sigqueue_cachep =
2568         kmem_cache_create("sigqueue",
2569             sizeof(struct sigqueue),
2570             __alignof__(struct sigqueue),
2571             0, NULL, NULL);
2572     if (!sigqueue_cachep)
2573         panic("signals_init(): cannot create sigqueue SLAB cache");
2574 }
```

第 2567~2571 行: 为 `sigqueue` 分配 SLAB 内存。

8.5.29 调用 `page_writeback_init()`

第 479 行: 函数 `page_writeback_init()` 初始化一些值, 当脏页写回磁盘时这些值起控制作用。脏页并不立即写回磁盘, 而是当一定时间后或内存中一定数目和比例的页面被标记为脏页时才写回磁盘。该 `init` 函数尝试确定在触发后台回写和专门的回写操作之前脏页的最佳值。后台脏页回写比专门的脏页回写花费的处理器时间要少得多。

```
-----
mm/page-writeback.c
488 /*
489 * If the machine has a large highmem:lowmem ratio then scale back the
490 * default
491 * dirty memory thresholds: allowing too much dirty highmem pins an excessive
492 * number of buffer_heads.
493 */
494 void __init page_writeback_init(void)
495 {
496     long buffer_pages = nr_free_buffer_pages();
497     long correction;
498     total_pages = nr_free_pagecache_pages();
499     correction = (100 * 4 * buffer_pages) / total_pages;
500     if (correction < 100) {
501         dirty_background_ratio *= correction;
502         dirty_background_ratio /= 100;
503         vm_dirty_ratio *= correction;
504         vm_dirty_ratio /= 100;
505     }
506     mod_timer(&wb_timer, jiffies + (dirty_writeback_centisecs * HZ) / 100);
507     set_ratelimit();
508 }
```

```

510 register_cpu_notifier(&ratelimit_nb);
511 }
-----

```

第 495~507 行: 如果正在操作的机器拥有的页缓存数量相对于缓冲区页面数量来说, 数量众多, 因而可以降低脏页回写的阈值。如果不降低该阈值而提高回写频率, 那么每次回写时就需要使用大量 buffer_heads。(这就是 page_writeback() 前面那个注释的意思。)

默认的后台回写 dirty_background_ratio 从有 10% 的脏页时开始。专门的回写 vm_dirty_ratio 从 40% 的页面为脏页时开始。

第 508 行: 修改回写定时器 wb_timer, 使之被周期性地触发 (在默认状态下, 每 5s 触发一次)。

第 509 行: 调用 set_ratelimit(), 该函数有极详细的文档说明。其内部注释如下。

```

-----
mm/page-writeback.c
450 /*
451 * If ratelimit_pages is too high then we can get into dirty-data overload
452 * if a large number of processes all perform writes at the same time.
453 * If it is too low then SMP machines will call the (expensive)
454 * get_writeback_state too often.
455 *
456 * Here we set ratelimit_pages to a level which ensures that when all CPUs
are
457 * dirtying in parallel, we cannot go more than 3% (1/32) over the dirty
memory
458 * thresholds before writeback cuts in.
459 *
460 * But the limit should not be set too high. Because it also controls the
461 * amount of memory which the balance_dirty_pages() caller has to write back.
462 * If this is too large then the caller will block on the IO queue all the
463 * time. So limit it to four megabytes - the balance_dirty_pages() caller
464 * will write six megabyte chunks, max.
465 */
466
467 static void set_ratelimit(void)
468 {
469     ratelimit_pages = total_pages / (num_online_cpus() * 32);
470     if (ratelimit_pages < 16)
471         ratelimit_pages = 16;
472     if (ratelimit_pages * PAGE_CACHE_SIZE > 4096 * 1024)
473         ratelimit_pages = (4096 * 1024) / PAGE_CACHE_SIZE;
474 }
-----

```

第 510 行: page_writeback_init() 最后的命令向 CPU 通告程序注册名为 ratelimit_nb 的速率限制通告程序块。被通告时, 这个通告程序块调用 ratelimit_handler(), 该函数再来调用 set_ratelimit(), 其目的在于, 当在线 CPU 的数目发生变化时, 重新计算 ratelimit_pages。

```

-----
mm/page-writeback.c

```

```

483 static struct notifier_block ratelimit_nb = {
484     .notifier_call = ratelimit_handler,
485     .next = NULL,
486 };

```

最后, 还需要检查当 `wb_timer` (从第 508 行开始) 离开并调用 `wb_time_fn()` 时究竟发生了什么:

```

-----
mm/page-writeback.c
414 static void wb_timer_fn(unsigned long unused)
415 {
416     if (pdflush_operation(wb_kupdate, 0) < 0)
417         mod_timer(&wb_timer, jiffies + HZ); /* delay 1 second */
418 }
-----

```

第416~417行: 当定时器响起时, 内核触发 `pdflush_operation()`, 该函数唤醒一个 `pdflush` 线程将实际的脏页写回磁盘。如果 `pdflush_operation()` 不能唤醒任何 `pdflush` 线程, 那么它将通知回写定时器在 1s 内再次触发, 试图唤醒一个 `pdflush` 线程。有关 `pdflush` 的详情参见第9章。

8.5.30 调用 `proc_root_init()`

第 480~482 行: 在第 2 章中, 我们已经解释过 `CONFIG_*` 是指编译时的变量。若在编译时选中了 `proc` 文件系统, 初始化的下一步操作就是调用 `proc_root_init()`。

```

-----
fs/proc/root.c
40 void __init proc_root_init(void)
41 {
42     int err = proc_init_inodecache();
43     if (err)
44         return;
45     err = register_filesystem(&proc_fs_type);
46     if (err)
47         return;
48     proc_mnt = kern_mount(&proc_fs_type);
49     err = PTR_ERR(proc_mnt);
50     if (IS_ERR(proc_mnt)) {
51         unregister_filesystem(&proc_fs_type);
52         return;
53     }
54     proc_misc_init();
55     proc_net = proc_mkdir("net", 0);
56 #ifdef CONFIG_SYSVIPC
57     proc_mkdir("sysvipc", 0);
58 #endif
59 #ifdef CONFIG_SYSCTL
60     proc_sys_root = proc_mkdir("sys", 0);
61 #endif

```

```

62  #if defined(CONFIG_BINFMT_MISC) || defined(CONFIG_BINFMT_MISC_MODULE)
63      proc_mkdir("sys/fs", 0);
64      proc_mkdir("sys/fs/binfmt_misc", 0);
65  #endif
66      proc_root_fs = proc_mkdir("fs", 0);
67      proc_root_driver = proc_mkdir("driver", 0);
68      proc_mkdir("fs/nfsd", 0); /* somewhere for the nfsd filesystem to be
        mounted */
69  #if defined(CONFIG_SUN_OPENPROMFS) || defined(CONFIG_SUN_OPENPROMFS_MODULE)
70      /* just give it a mountpoint */
71      proc_mkdir("openprom", 0);
72  #endif
73      proc_tty_init();
74  #ifdef CONFIG_PROC_DEVICETREE
75      proc_device_tree_init();
76  #endif
77      proc_bus = proc_mkdir("bus", 0);
78  }

```

第 42 行: 本行初始化为索引节点缓存, 用于保存该文件系统的索引节点。

第 45 行: 向内核注册 `file_system_type` 类型的结构 `proc_fs_type`。我们来认真分析这一结构。

```

-----
fs/proc/root.c
33  static struct file_system_type proc_fs_type = {
34      .name      = "proc",
35      .get_sb     = proc_get_sb,
36      .kill_sb    = kill_anon_super,
37  };

```

`file_system_type` 结构简单地将文件系统的名字定义为 `proc`, 该结构包含检索和释放超级块结构的例程。

第 48 行: 挂载文件系统 `proc`。欲知此处发生了什么, 详情可查阅 `kern_mount` 的附加内容。

第 54~78 行: 调用 `proc_misc_init()` 来创建 `/proc` 文件系统中绝大多数的项。它通过调用 `create_proc_read_entry()`、`create_proc_entry()` 和 `create_proc_seq_entry()` 来创建项。该代码块的剩余部分由 `proc_mkdir` 调用和 `proc_tty_init()` 例程调用组成, 它们分别用于创建 `/proc/` 下的目录和 `/proc/tty` 下的树。如果设置了 `CONFIG_PROC_DEVICETREE` 的配置时间值, 那么, 还要调用 `proc_device_tree_init()` 例程来创建 `/proc/device-tree` 子树。

8.5.31 调用 `init_idle()`

第 490 行: `start_kernel()` 快结束时使用参数 `current` 来调用 `init_idle()`, 而 `smp_processor_id()` 则准备重新调度 `start_kernel()`。

```

-----
kernel/sched.c
2643 void __init init_idle(task_t *idle, int cpu)

```



```

2644 {
2645     runqueue_t *idle_rq = cpu_rq(cpu), *rq = cpu_rq(task_cpu(idle));
2646     unsigned long flags;
2647
2648     local_irq_save(flags);
2649     double_rq_lock(idle_rq, rq);
2650
2651     idle_rq->curr = idle_rq->idle = idle;
2652     deactivate_task(idle, rq);
2653     idle->array = NULL;
2654     idle->prio = MAX_PRIO;
2655     idle->state = TASK_RUNNING;
2656     set_task_cpu(idle, cpu);
2657     double_rq_unlock(idle_rq, rq);
2658     set_tsk_need_resched(idle);
2659     local_irq_restore(flags);
2660
2661     /* Set the preempt count _outside_ the spinlocks! */
2662     #ifdef CONFIG_PREEMPT
2663         idle->thread_info->preempt_count = (idle->lock_depth >= 0);
2664     #else
2665         idle->thread_info->preempt_count = 0;
2666     #endif
2667 }

```

第 2645 行：存储当前所在 CPU 的请求队列以及给定任务 `idle` 所在 CPU 的请求队列。本例中使用 `current` 和 `smp_processor_id()`，这些请求队列完全相等。

第 2648~2649 行：保存 IRQ 标志并获得这两个请求队列锁。

第 2651 行：设置运行 `idle` 任务的 CPU 请求队列的当前任务。

第 2652~2656 行：这些语句将任务 `idle` 从请求队列中移出，并移入 CPU 请求队列 `cpu` 中。

第 2657~2659 行：释放之前锁定的请求队列的锁。然后，将任务 `idle` 标记为重新调度，并恢复先前保存的 IRQ。最后，如果配置了内核抢占的话，则设置抢占计数器。

8.5.32 调用 `rest_init()`

第 493 行：例程 `rest_init()` 相当简单。事实上，它只创建一个称为 `init` 的线程，同时删除初始化内核锁，并调用 `idle` 线程：

```

-----
init/main.c
388 static void noinline rest_init(void)
389 {
390     kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);
391     unlock_kernel();
392     cpu_idle();
393 }
-----

```

第 388 行：读者可能已经注意到了，调用的第一个例程是 `start_kernel()` 而不是 `__init`。

回想一下，在第 2 章，我们说过当一个函数前加了 `__init` 时，表示用来保存函数变量等内容的内存，在初始化过程接近尾声时，这些内存就会被清除或释放。这可以通过调用 `free_initmem()` 来实现，分析 `init()` 时我们见过这个函数。`rest_init()` 不是 `__init` 函数的原因在于，它在完成前调用了 `init` 线程（即调用 `cpu_idle`）。因为 `init` 线程执行 `free_initmem()` 调用，所以可能存在竞态条件，这就是 `free_initmem()` 要在 `rest_init()`（或根线程）完成前被调用的原因。

第 390 行：本行创建 `init` 线程，通常也叫 `init` 进程或进程 1。简单地说，此处该线程和调用进程共享所有的内核数据结构。内核线程调用 `init()` 函数，在 8.6 节将讨论该函数。

第 391 行：当仅有单处理器时，例程 `unlock_kernel()` 什么都不做。否则，该函数释放大内核锁 BKL。

第 392 行：调用 `cpu_idle()` 将根线程转换为空闲线程。该例程将处理器让给调度程序，当调度程序没有其他挂起的进程可投入运行时，就返回该例程。

至此，已经完成了 Linux 内核初始化的大部分工作。下面，简要探讨 `init()` 函数。

8.6 init 线程（或进程 1）

现在，来看看 `init` 线程。注意，为简单起见，我们略过了所有与 SMP 相关的例程：

```
-----
init/main.c
601 static int init(void * unused)
602 {
603     lock_kernel();
...
612     child_reaper = current;
...
627     populate_rootfs();

629     do_basic_setup();
...
635     if (sys_access((const char __user *) "/init", 0) == 0)
636         execute_command = "/init";
637     else
638         prepare_namespace();
...
645     free_initmem();
646     unlock_kernel();
647     system_state = SYSTEM_RUNNING;

649     if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
650         printk("Warning: unable to open an initial console.\n");
651
652     (void) sys_dup(0);
653     (void) sys_dup(0);
...
662     if (execute_command)
663         run_init_process(execute_command);
```

```

664
665  run_init_process("/sbin/init");
666  run_init_process("/etc/init");
667  run_init_process("/bin/init");
668  run_init_process("/bin/sh");
669
670  panic("No init found. Try passing init= option to kernel.");
671  }

```

第 612 行: `init` 线程收养其父进程已经死亡的任何线程。变量 `child_reaper` 是一个全局指针, 在 `init/main.c` 中定义, 指向一个 `task_struct`。该变量准备承担“养父”的职责, 它所指向的线程将会成为新的父进程。提到诸如 `reparent_to_init()` (详细内容请查阅 `kernel/exit.c`), `choose_new_parent()` (详细内容请查阅 `kernel/exit.c`) 和 `forget_original_parent()` (详细内容请查阅 `kernel/exit.c`) 等函数是因为它们使用 `child_reaper` 来重置调用线程的父进程。

第 629 行: 函数 `do_basic_setup()` 用于初始化驱动程序模型, `sysctl` 接口, 网络套接字接口以及所支持的工作队列。

```

-----
init/main.c
551  static void __init do_basic_setup(void)
552  {
553      driver_init();
554
555      #ifdef CONFIG_SYSCTL
556          sysctl_init();
557      #endif
558
559      ...
560      sock_init();
561
562      init_workqueues();
563      do_initcalls();
564  }

```

第 553 行: 函数 `driver_init()` (详细内容请查阅 `drivers/base/init.c`) 初始化与驱动程序相关的所有子系统。这是设备驱动程序初始化的第一部分。第二部分见第 563 行的函数调用 `do_initcalls()`。

第 555~557 行: `sysctl` 接口为内核参数的动态转换提供支持。这意味着可以在运行时修改 `sysctl` 支持的内核参数, 而不需要重新编译和引导内核。`sysctl_init()` (详细内容请查阅 `kernel/sysctl.c`) 初始化该接口。有关 `sysctl` 的详情可参阅手册 (`man sysctl`)。

第 560 行: 函数 `sock_init()` 是个伪函数, 如果内核不支持网络的话, 该函数仅有 `printk` 语句, 此时, 该函数就在 `net/nonet.c` 中定义。如果内核支持网络功能, 则 `sock_init()` 在 `net/socket.c` 中定义, 作用是初始化用于支持网络功能的存储器高速缓存, 并注册支持网络的文件系统。

第 562 行: `init_workqueues` 设置工作队列通告程序链。第 10 章将讨论工作队列。

第 563 行: 函数 `do_initcalls()` (详细内容请查阅 `init/main.c`) 构成了设备驱动程序初始化的第二部分。该函数接着调用函数指针数组中的元素, 这些函数指针对应着内置的设备初始化

函数^①。

第 635~638 行：如果存在早期的用户空间 `init`，则内核并不准备命名空间，它允许执行这个函数。否则，调用函数 `prepare_namespace()`。命名空间指的是文件系统层次结构的挂载点。

```
-----
init/do_mounts.c
383 void __init prepare_namespace(void)
384 {
385     int is_floppy;
386
387     mount_devfs();
388
389     ...
391     if (saved_root_name[0]) {
392         root_device_name = saved_root_name;
393         ROOT_DEV = name_to_dev_t(root_device_name);
394         if (strncmp(root_device_name, "/dev/", 5) == 0)
395             root_device_name += 5;
396     }
397
398     is_floppy = MAJOR(ROOT_DEV) == FLOPPY_MAJOR;
399
400     if (initrd_load())
401         goto out;
402
403     if (is_floppy && rd_doload && rd_load_disk(0))
404         ROOT_DEV = Root_RAM0;
405
406     mount_root();
407 out:
408     umount_devfs("/dev");
409     sys_mount(".", "/", NULL, MS_MOVE, NULL);
410     sys_chroot(".");
411     security_sb_post_mountroot();
412     mount_devfs_fs ();
413 }
-----
```

第 387 行：函数 `mount_devfs()` 创建与挂载相关的结构 `/dev`。我们需要挂载 `/dev`，因为要使用它来查阅根设备的名称。

第 391~396 行：该代码块设置全局变量 `ROOT_DEV`，用来表示通过内核引导时参数传递进来的根设备。

第 398 行：主设备号的简单比较，以说明该根设备是不是软盘。

第 400~401 行：如果 RAM 盘是内核的根文件系统，则调用 `initrd_load()` 来挂载 RAM 盘。如果是这样的话，该函数返回 1 并跳转到标记 `out` 处，它将撤销来自设备的根文件系统准备过程中所作的一切工作。

第 406 行：调用 `mount_root` 来完成挂载根文件系统的主要工作。接下来深入分析一下该

① 关于 Trevor Woerner 对 `initcall` 机制更多精辟的论述可参阅 <http://geek.vtnet.ca/doc/initcall/>。

函数：

```
-----
init/do_mounts.c
353 void __init mount_root(void)
354 {
355 #ifdef CONFIG_ROOT_NFS
356   if (MAJOR(ROOT_DEV) == UNNAMED_MAJOR) {
357     if (mount_nfs_root())
358       return;
359
360     printk(KERN_ERR "VFS: Unable to mount root fs via NFS, trying
floppy.\n");
361     ROOT_DEV = Root_FD0;
362   }
363 #endif
364 #ifdef CONFIG_BLK_DEV_FD
365   if (MAJOR(ROOT_DEV) == FLOPPY_MAJOR) {
...
367     if (rd_doload==2) {
368       if (rd_load_disk(1)) {
369         ROOT_DEV = Root_RAM1;
370         root_device_name = NULL;
371       }
372     } else
373       change_floppy("root floppy");
374   }
375 #endif
376   create_dev("/dev/root", ROOT_DEV, root_device_name);
377   mount_block_root("/dev/root", root_mountflags);
378 }
-----
```

第 355~358 行：如果内核已挂载了 NFS 文件系统，则应该执行 `mount_nfs_root()`。如果挂载 NFS 失败，那么内核应输出适当的信息，并设法挂载软盘作为根文件系统。

第 364~375 行：在该代码块中，内核设法挂载根软盘^①。

第 377 行：该函数执行挂载根设备时的大部分工作。现在，让我们回到 `init()`。

第 645 行：调用 `free_initmem()`，释放那些带 `__init` 前缀的例程用完的所有内存段。这标志着退出纯内核空间，并开始设置用户模式的数据。

第 649~650 行：打开初始控制台。

第 662~668 行：变量 `execute_command` 在 `init_setup()` 中设置并保存引导参数的值，若不希望调用默认的 `/sbin/init` 的话，该参数包含调用 `init` 程序的名字。若传递了 `init` 程序的名字，该程序将获得比普通 `/sbin/init` 高的优先级。要注意的是，由于最后要调用 `execve()`，所以 `run_init_process()`（详细内容请查阅 `init/main.c`）并不返回值。因此，最初成功运行的 `init` 函数就是唯一运行的函数。如果没有找到 `init` 程序，可以使用 `bash shell` 来启动。

① 要注意 `rd_doload`：如果没有加载 RAM 盘，那么该全局变量的值为 0；如果已加载 RAM 盘，则其值为 1；“双 `initrd/ramload` 设置”时，其值为 2。

第 670 行：如果设法执行各种 `init` 程序的所有办法都失败了，那么将执行该错误语句。

内核的初始化到此为止。从现在开始，`init` 进程本身涉及系统的初始化，并启动所有必需的进程以及用户登录所需的后台支持。

8.7 小结

本章描述系统加电和内核引导期间发生了什么事情。我们讨论了 BIOS 和 Open Firmware，以及它们是如何与内核引导加载程序交互的。接下来讨论了几个最常用的引导加载程序 LILO、GRUB 和 Yaboot，并简单概括了它们如何工作，以及如何调用第一个内核初始化例程。

同时，我们也分析了建立内核初始化过程的函数。在整个初始化过程中，分析了内核代码，也谈到了前几章介绍的概念。特别是通过以下高级操作跟踪了 Linux 的内核初始化：

- ☐ 启动和锁住内核；
- ☐ 为 Linux 的内存管理初始化页高速缓存和页面地址；
- ☐ 为多 CPU 做好准备；
- ☐ 显示内核标志；
- ☐ 初始化 Linux 调度程序；
- ☐ 分析传到 Linux 内核的参数；
- ☐ 初始化中断处理程序、定时器处理程序和信号处理程序；
- ☐ 挂载初始文件系统；
- ☐ 完成系统的初始化，并将控制权从 `init` 交回系统。

离开内核初始化过程时，我们必须提到，内核还是在起作用的，它首先启动许多高级 Linux 应用程序，例如 X11、sendmail 等。所有这些程序都依赖于我们刚探讨过的基本结构和设置。

8.8 习题

1. 大内核锁 (BLK) 和一般自旋锁之间有什么区别？
2. `init` 脚本允许向 Linux 内核加入什么特殊的安全特性？
3. 谁为内核的页面管理初始化数据结构？
4. 触发后台程序将脏页写回磁盘时，脏页所占的百分比是多少？触发专门的写回程序时，脏页所占的百分比是多少？
5. 为什么是 `rest_init()` 不是 `__init` 函数？

第9章

构建 Linux 内核

本章内容

- 工具链
- 内核源代码的构建

到此为止，我们已经认识了 Linux 内核中的子系统，也探究了系统的初始化函数。同样，了解内核映像的创建也是非常重要的。本章将讨论内核映像的编译和链接过程，并考察内核的构建过程。

9.1 工具链

工具链 (toolchain) 是创建 Linux 内核映像的一组程序的集合。链的概念源于一个工具的输出将作为另一工具的输入这一特点。工具链包含编译程序、汇编程序和链接程序。从技术上讲，还需一个文本编辑器。本节只讨论前三个工具。开发软件时工具链必不可少，当然，SDK (软件开发工具包) 也是必需的。

编译程序 (compiler) 是一个转换程序，可将高级源语言转换成低级目标语言 (object language)。目标代码是运行在目标系统上并依赖于机器的一系列指令。**汇编程序** (assembler) 也是一个转换程序，可将汇编语言转换成与编译程序相同的目标代码。其不同之处在于：汇编程序产生的机器指令与汇编语句一一对应，而高级语言的每条语句可能对应多条机器指令。如你所知，Linux 源代码中与体系结构相关部分的某些文件是用汇编语言编写的，它们调用汇编程序将其编译为目标代码。

链接编辑程序 (link editor) 或**链接程序** (linker) 可将多个可执行模块链接成一个单元，以便执行。

图 9-1 说明了工具链的“链式关系”。链接程序将把程序的目标代码与它使用的所有函数库链接起来。编译程序中含有标志位，可供用户选择编译级别。例如，从图 9-1 中可以看到，编译程序可以直接产生机器代码，也可以先编译成汇编源代码，再经由汇编程序把它组合成计算机可直接执行的机器指令。

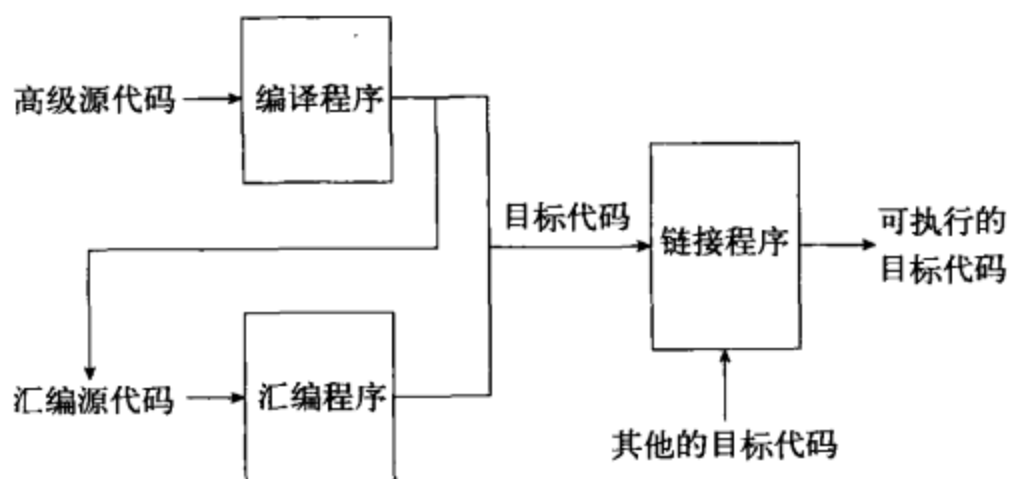


图 9-1 工具链

9.1.1 编译程序

常见的编译程序也具有链式特性，它分成多步执行，一步的输出作为下一步的输入，图 9-2 是其图解。编译的第一步是词法扫描 (scanner)，它将高级语言编写的程序拆分为一个个的单词符号 (tokens)。接下来，语法分析 (parser) 阶段根据一定的语法规则将这些符号组织起来，并由上下文分析阶段根据语义属性进一步组织这些符号。优化程序可提高所解析的符号的效率，代码生成阶段则产生目标代码。编译程序输出符号表和可重定位的目标代码。换句话说，每一个已编译的模块其起始地址都是 0，链接时必须重定位到合适的位置。

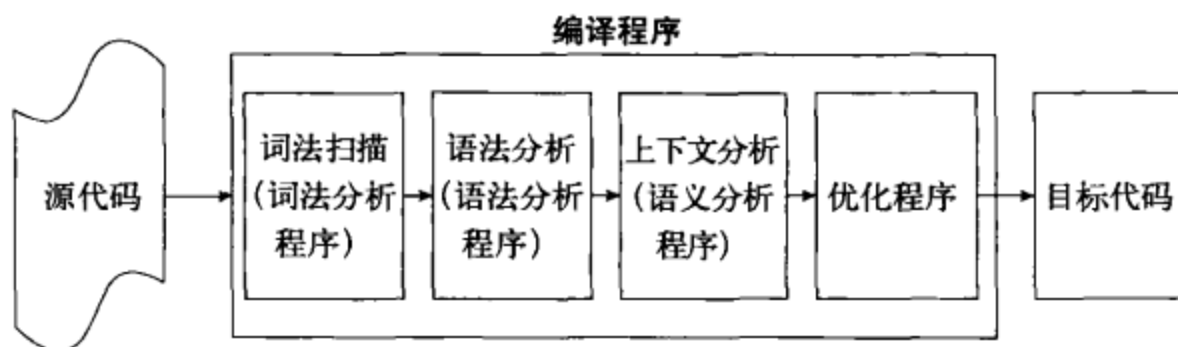


图 9-2 编译程序的操作过程

9.1.2 交叉编译

工具包通常自然地运行，这就意味着编译代码的系统和运行工具包生成的目标代码的系统相同。如果在 x86 系统上开发了一个内核，把它加载到另一个（或同样的）x86 系统上时，无须进行任何编译工作。功能强大的 Mac 和数量繁多的 x86 工具包可将代码编译成能在相应体系结构上运行的代码。但是，如果我们想在一种平台下编写代码，而在另一种平台下运行呢？

听来容易做起来难！考虑一下嵌入式市场，嵌入式系统通常是在有限内存和 I/O 接口的环境中实现其功能。无论控制的是汽车、路由器还是手机，这些设备都不足以容纳一个完整的嵌入式系统开发环境（更不用说监视器和键盘了）。解决的办法就是，让开发者利用他们功能强大、价格相对廉价的工作站作为宿主机系统 (host system) 来开发代码，然后将这些代码下载到目标系统 (target system) 上进行检测。因而称之为交叉编译。

例如，你可能是 PowerPC 嵌入式系统的开发者，该系统使用 405 处理器。而你的台式机的开

发系统绝大多数都是 x86 的。使用 gcc，你可以在这些台式机上完成所有开发工作（包括 C 程序和汇编程序），只要在编译时使用 `-mcpu=405` 选项^①就可以了。这样产生的目标代码具有 405 处理器特有的指令和寻址方式。之后就可以把可执行的目标代码下载到嵌入式系统中运行并调试。这样做看起来确实很乏味，但是对资源有限的嵌入式系统而言，它大大节省了内存。

在这种特殊环境下，许多针对交叉编译的嵌入式代码的辅助开发调试工具也应运而生。

9.1.3 链接程序

编译一个 C 程序时（例如，“Hello world!”），生成的代码远不止.c 文件中的这三四行。链接程序的工作就是找出所有引用的外部模块并链接起来。这些外部模块或函数库一般来自于开发者、操作系统和 C 运行库（`printf()` 的出处）。链接程序取出这些函数库，修订指针位置（**重定位**），并交叉引用模块中的**符号解析**，最终产生一个可执行模块。符号可以是全局的也可以是局部的。全局符号可以在模块内部定义，或由另一模块外部引用。链接程序要找到与模块相关的所有符号的定义。（注意，内核开发者无法使用用户空间库。）对于普通函数，内核自身有对应的函数。**静态库**是在链接时被找到并复制的，而**动态库**和**共享库**是在运行时才装载的，并让所有的进程共享。Microsoft 和 OS/2 将共享库称为动态链接库。Linux 提供的系统调用 `dlopen()`、`dlsym()` 及 `dlclose()` 用于加载/打开共享库，查找库中的符号，然后关闭共享库。

9.1.4 ELF 二进制目标文件

目标文件的格式随着生产商的不同而不同。现今，大多数 UNIX 系统都使用可执行链接格式 (ELF)。ELF 文件种类繁多，每种文件都执行不同的功能。主要的 ELF 文件类型有可执行文件、可重定位目标文件、核心文件及共享库。ELF 格式支持目标文件对不同平台和体系结构兼容。图 9-3 阐明可执行 ELF 目标文件和非可执行 ELF 目标文件。

非可执行 ELF 目标文件	可执行 ELF 目标文件
ELF 头	ELF 头
程序头表（可选）	程序头表（用于加载的节）
第 1 节	第 1 节
第 2 节	第 2 节
第 3 节	第 3 节
...	...
节头表 （用于链接的节）	节头表（可选）

图 9-3 可执行 ELF 文件和非可执行 ELF 文件

^① 要了解更多应用于 IBM RS/6000(POWER)及 PowerPC 的 gcc 选项，可登录 http://gcc.gnu.org/onlinedocs/gcc/RS_002f6000-and-PowerPC-Options.html#RS_002f6000-and-PowerPC-Options。

ELF 文件头总是位于 ELF 文件中偏移为 0 的位置，ELF 文件的所有信息均可通过 ELF 文件头获得。ELF 文件头是目标文件中唯一固定的结构，因而它必须能够定位并说明文件中其他子结构的位置和大小。所有 ELF 文件都被分为许多具有相似数据块的节 (section) 或段 (segment)。非可执行目标文件包含多个节和一个节头表 (section header table)，可执行目标文件则包含多个段和一个程序头表 (program header table)。

1. ELF 文件头

在 linux 系统中，数据结构 `elf32_hdr` 存储 ELF 文件头 (32 位的系统使用该结构，64 位的系统则使用 `elf64_hdr` 结构)。我们来看看这个结构：

```
-----
include/linux/elf.h
234 #define EI_NIDENT 16
235
236 typedef struct elf32_hdr{
237     unsigned char  e_ident[EI_NIDENT];
238     Elf32_Half     e_type;
239     Elf32_Half     e_machine;
240     Elf32_Word     e_version;
241     Elf32_Addr     e_entry; /* Entry point */
242     Elf32_Off      e_phoff;
243     Elf32_Off      e_shoff;
244     Elf32_Word     e_flags;
245     Elf32_Half     e_ehsize;
246     Elf32_Half     e_phentsize;
247     Elf32_Half     e_phnum;
248     Elf32_Half     e_shentsize;
249     Elf32_Half     e_shnum;
250     Elf32_Half     e_shstrndx;
251 } Elf32_Ehdr;
-----
```

第 237 行：e_ident 字段存放了一个 16 字节的魔数，用于标识该文件是否为 ELF 文件。

第 238 行：e_type 字段指定目标文件类型，例如，可执行文件、重定位文件、共享的目标文件等。

第 239 行：e_machine 字段表示被编译文件所在系统的体系结构。

第 240 行：e_version 字段表示目标文件的版本。

第 241 行：e_entry 字段保存程序的起始地址。

第 242 行：e_phoff 字段保存程序头表在文件中的偏移量（按字节计数）。

第 243 行：e_shoff 字段保存节头表在文件中的偏移量（按字节计数）。

第 244 行：e_flags 字段保存特定于处理器的标志。

第 245 行：e_ehsize 字段保存 ELF 头的大小。

第 246 行：e_phentsize 字段保存程序头表中每一项的大小。

第 247 行：e_phnum 字段包含程序头中表项的个数。

第 248 行：e_shentsize 字段保存节头表中每一项的大小。

第 249 行：e_shnum 字段保存节头中项的数量，表明该文件有多少节。

第 250 行：e_shstrndx 字段保存节头中节字符串的索引。

2. 节头表

节头表是 `Elf32_Shdr` 类型的数组，它在 ELF 文件中的偏移由 ELF 头的 `e_shoff` 字段来确定。在 ELF 文件中，每节都有一个对应的节头表：

```
-----
include/linux/elf.h
332  typedef struct {
333      Elf32_Word  sh_name;
334      Elf32_Word  sh_type;
335      Elf32_Word  sh_flags;
336      Elf32_Addr  sh_addr;
337      Elf32_Off   sh_offset;
338      Elf32_Word  sh_size;
339      Elf32_Word  sh_link;
340      Elf32_Word  sh_info;
341      Elf32_Word  sh_addralign;
342      Elf32_Word  sh_entsize;
343  } Elf32_Shdr;
-----
```

第 333 行：sh_name 字段包含节名。

第 334 行：sh_type 字段包含节的内容。

第 335 行：sh_flags 字段包含各种属性的信息。

第 336 行：sh_addr 字段保存节在内存映像中的地址。

第 337 行：sh_offset 字段保存 ELF 文件中这一节中初始字节的偏移量。

第 338 行：sh_size 字段包含节的大小。

第 339 行：sh_link 字段包含表链接 (table link) 的索引，该索引视其 sh_type 值而定。

第 340 行：sh_info 字段包含附加信息，视其 sh_type 值而定。

第 341 行：sh_addralign 字段包含地址对齐的约束。

第 342 行：当节中保存一张固定大小的表时，sh_entsize 字段包含节中每项的大小。

3. 非可执行 ELF 文件的节

ELF 文件被分成了多个节，每节都包含特定类型的信息。表 9-1 列出了这些节的信息类型，只有当在编译时设置了一定的编译器标志位，某些节才会出现。回忆一下，`ELF32_Ehdr->e_shnum` 保存了 ELF 文件中的节数。

表9-1 ELF文件节

节 名	说 明
.bss	未初始化的数据
.comment	GCC用于表示编译器版本的信息
.data	已初始化的数据
.debug	以符号表的形式给出的符号调试信息
.dynamic	动态链接信息
.dynstr	动态链接时的字符串
.fini	进程的终止代码，GCC的退出代码

(续)

节 名	说 明
.got	全局偏移量表
.hash	符号散列表
.init	初始化代码
.interp	程序解释器的路径名
.line	标记调试时的行号
.note	编译程序控制版本时所需的信息
.plt	过程链接表
.relname	重定位信息
.rodata	只读数据
.shstrtab	节名
.symtab	符号表
.text	可执行的指令

4. 程序头表

可执行的或共享的目标文件其程序头表是一个结构体数组。每个结构体描述一个段或执行该程序所需的其他信息：

```

-----
include/linux/elf.h
276 typedef struct elf32_phdr{
277     Elf32_Word  p_type;
278     Elf32_Off   p_offset;
279     Elf32_Addr  p_vaddr;
280     Elf32_Addr  p_paddr;
281     Elf32_Word  p_filesz;
282     Elf32_Word  p_memsz;
283     Elf32_Word  p_flags;
284     Elf32_Word  p_align;
285 } Elf32_Phdr;
-----

```

第 277 行：p_type 字段描述该段的类型。

第 278 行：p_offset 字段保存该段的开始相对于文件开始的偏移量。

第 279 行：若使用了该段，则 p_vaddr 字段保存该段的虚拟地址。

第 280 行：若使用了该字段，则 p_paddr 字段保存该段的物理地址。

第 281 行：p_filesz 字段保存文件映像中该段的字节数。

第 282 行：p_memsz 字段保存内存映像中该段的字节数。

第 283 行：p_flags 字段保存的标志依 p_type 而定。

第 284 行：p_align 字段描述要对齐的段在内存中如何对齐。该值是 2 的整数次幂。

通过这些信息，系统函数 exec() 和链接程序合作，为可执行程序在内存中创建进程映像，该过程包含如下步骤：

- 将可执行文件的段加入内存；
- 加载所有需要的共享库；
- 需要时重定位可执行文件及其共享对象；
- 将控制权交给程序；

理解目标文件的格式及可用的开发工具，有利于更好地调试编译时出现的问题（例如，未知的引用）。同时，了解代码载入和重定位之后的位置，有利于解决运行时出现的问题。

9.2 内核源代码的构建

现在，我们来讨论如何把内核编译成二进制映像文件，以及如何在执行前载入内存。作为一个内核开发者，你将会与源代码结下不解之缘，因而，有必要掌握如何阅读分析源代码以及如何编辑构建系统，以便添加自己的代码。

本节将带领你从下载源代码开始，直至教会你如何编译刚加载的内核映像文件。其中会谈到内核映像的创建，但不会是详细的、一步一步的指令手册式说明，网上许多关于如何构建内核映像的文档都很全面，例如内核 HOWTO (www.tldp.org/HOWTO/Kernel-HOWTO/)，该文档目前还在不断更新。它的目的是为将修改合并到构建系统，提供你所需要的信息。

程序开发者对构建系统或 Makefiles 并没有极大兴趣，但我们同样需要理解内核构建系统，以及如何通过修改源代码来更新内核。Linux 2.6 版的内核中又增添了许多工具帮助你更好地理解与内核构建系统有关的所有选项。同样，构建系统也经过较大地调整并重新设计，使之更有效地用文档记录。

本节将介绍如何布局源代码、如何构建内核以及 Makefiles 如何工作。第一步是获取源代码。我们先来看看源代码的结构，以及从何处获取源代码。

9.2.1 解释源代码

Linux 的官方源代码发布网址是 www.kernel.org。可下载的源代码有 gzip 压缩的.tar.gz 包和 bzip2 压缩的.tar.bz2 包。这些压缩包的源代码适用于任何体系结构。

内核开发者修改内核源代码后，必须将改动后的源代码提交给内核维护人员，由内核维护人员决定是否将这些改动加入下一个稳定树，最新的 PPC 开发通常在 www.penguinppc.org 的一棵独立的树下维护，所有改动都被保存在 PPC 树之下，当认为这些改动稳定而有价值时，就会被收录到主树中。目前，Linux 的 PPC 社团正在朝着主树进军。

源代码的位置在一定程度上取决于其发布版。例如，在红帽系统中，源代码被放置在 `/usr/src/linux-<version>/` 目录下（要么默认安装，要么被 RPM 安装）。若进行交叉编译，就是说，为某一体系结构构建的内核与进行实际编译的体系结构不同，源代码可能位于宿主机的某个 `/opt/<distribution name>/` 目录下，或在用户 chroot 到的根文件系统映像下。例如，面向嵌入式 Linux 市场的 Montavista 发布版，其源代码（和交叉编译程序）默认存储在 `/opt/mvista/` 目录下。

本节将源代码文件系统的根目录仅叫做根，在红帽发布版中，源代码的根目录位于 `/usr/src/linux-<version>/` 下。图 9-4 详细解释了源代码的结构化布局。

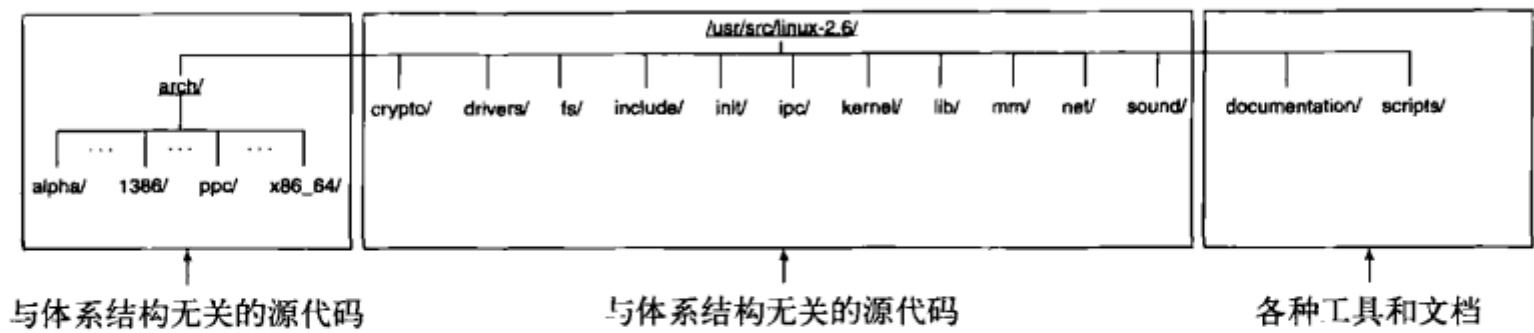


图 9-4 源代码的布局

源代码分为与体系结构相关的部分和与体系结构无关的部分。根目录下的 arch/ 目录包含了所有与体系结构相关的代码。该子目录下列出了从 kernel.org 镜像下载的源代码支持的所有体系结构。每一种支持的体系结构在 arch/ 下有一个相应的目录，其中包含与该体系结构相关代码的进一步细分。图 9-5 通过显示 arch/ 目录下的列表来展示它所支持的体系结构。

```
[1kp@ulaanbataar linux-2.6.7]$ ls arch/
alpha  arm26  h8300  ia64   m68knommu  parisc  ppc64  sh      sparc64  v850
arm    cris   i386   m68k   mips        ppc     s390   sparc   um       x86_64
[1kp@ulaanbataar linux-2.6.7]$
```

图 9-5 ls/usr/src/linux/arch

我们首先考察与结构体系无关的源代码，以便于对源代码的组织情况有个大致的了解，然后概览与体系结构相关的源代码，最后简要介绍不属于任何一类的其他各式各样的文件。

1. 与体系结构无关的代码

与体系结构无关的源代码根据其功能划分为 11 个不同的子目录。表 9-2 列出了这些子目录。

表9-2 与体系结构无关的子目录

子 目 录	目录说明
crypto	保存加密的API及各种加密/解密算法的代码
drivers	设备驱动程序的代码
fs	VFS及Linux支持的所有文件系统的代码
include	头文件代码。该目录下存放的子目录大都以前缀asm开头，用于存储特定于体系结构的头文件，其余子目录存放与体系结构无关的头文件
init	引导程序代码以及初始化代码中与体系结构无关的部分
ipc	支持进程间通信（IPC）的代码
kernel	内核空间特有的代码
lib	辅助函数的代码
mm	用于内存管理的代码
net	支持各种网络协议的代码
sound	支持音频系统的代码

前面多个章节中已经探究了其中的一个或多个子目录中的源代码。从上下文的角度考虑,以下各节仅简单回顾某些子目录,而略过那些未详细讨论过的子目录。

- fs/

fs/目录进一步分为支持VFS的C源文件和 supports 的各种文件系统的子目录。正如第7章中介绍的那样,VFS是对各种不同类型文件系统的抽象层,每个子目录下的代码将在存储设备与VFS抽象层之间架起一座桥梁。

- init/

init/目录中包含系统初始化时所有必需的代码。执行这些代码时,所有内核子系统都会被初始化,并创建所有初始进程。

- kernel/

大多数与结构体系无关的内核代码以及大部分内核子系统的代码都保存在kernel/目录下。某些代码有属于自己的目录,这些目录和kernel/目录处于同一层,例如文件系统和内存,根据文件名就可以知道它所包含的代码。

- mm/

mm/目录下保存用于内存管理的代码,在第4章已经见过这些代码的例子。

2. 与体系结构相关的代码

与体系结构相关的代码是指与硬件直接打交道的内核源代码。在你苦苦钻研这部分代码时,我们要提醒你的是Linux最初是为x86开发的。为了降低移植的难度,变量名和全局内核结构照搬了大多数x86核心术语。如果你浏览PPC代码时,遇到了PPC中不存在的与地址转换模式有关的名字时,不用大惊小怪。

当我们列出arch/i386/及arch/ppc目录下的文件时,你会发现有三个相同的文件: defconfig、Kconfig 和 Makefile。这三个文件都是与内核构建系统的基础结构有关的。9.2.2节将介绍这三个文件的作用。

表9-3列出了arch/ppc目录下的文件和子目录并进行了简单的介绍,如果你已经了解了Makefiles和Kconfig文件的结构,将有利于你浏览每个子目录中的这些文件,并有利于你熟悉代码的位置。

表9-3 arch/ppc/目录下的源码列表

子目录	目录说明
4xx_io	MPC4xx特有的I/O部分的源代码,特别是IBM STB3xxx SICC串口
8260_io	用于MPC8260-communication选项的源代码
8xx_io	用于MPC8xx-communication选项的源代码
amiga	用于PowerPC-equipped Amiga计算机的源代码
boot	与PPC引导程序相关的源代码。该目录下还包含一个images子目录,用于存储已编译的可引导映像
config	用于构建特定PPC平台和体系结构的配置文件
kernel	与内核子系统硬件相关的源代码
lib	PPC特殊库文件的源代码

(续)

子 目 录	目录说明
math-emu	PPC数学仿真器的源代码
mm	内存管理中特定于PPC部分的源代码，可参阅第6章的详细讨论
platforms	特定于安装PPC芯片的平台的源代码
syslib	特定于通用硬件的子系统的部分源代码
xmon	特定于PPC调试器的源代码

arch/x86 下的目录结构和与 PPC 体系结构相关的目录结构类似，表 9-4 概括了它所包含的各种子目录。

表9-4 arch/x86源码列表

子 目 录	目录说明
boot	与x86引导及安装过程相关的源代码
kernel	与内核子系统硬件相关的源代码
lib	x86库文件的源代码
mach-x	x86子结构的源代码
math-emu	x86数学仿真函数的源代码
mm	x86中内存管理部分的源代码，可参阅第6章的详细讨论
oprofile	oprofile内核分析工具的源代码
pci	x86 PCI驱动程序
power	x86电源管理的源代码

你也许会奇怪为什么两种体系结构的源码列表大相径庭，原因在于功能划分的差异，在一种体系结构上的划分不一定适合另一体系结构。例如，在 PPC 中，PCI 驱动程序会随着平台和子结构的不同而不同，要想在 PPC 平台下建立一个简单 PCI 的子目录，就要比在 x86 平台下困难得多。

3. 文件和目录杂项

在源根目录中的一些文件，既和与体系结构相关的代码无关，又和与体系结构无关的代码无关，表 9-5 列出了这些文件。

表9-5 其他文件

文件/目录	说 明
COPYING	Linux的GPL许可证
CREDITS	列出了对Linux做出贡献的人名
MAINTAINERS	当提交内核改动时涉及的维护人员及说明
README	发布说明
REPORTING-BUGS	描述报告bug的过程
documentation/ scripts/	包含了Linux内核及其源代码的有关文档，当内核有所更新时，这里含有丰富的改动信息 构建内核的过程中可以使用的功能及脚本

9.2.2 构建内核映像

内核构建系统或者说 kbuild, 是一种在构建内核时, 可以对内核配置选项进行选择的机制。2.6 内核树中已经更新了这种机制, 新版本的 kbuild 不仅高速而且备有更完善的文档。kbuild 系统完全依赖于源代码的层次结构。

1. 内核配置工具

利用内核配置工具自动生成名为 .config 的内核配置文件, 这是构建内核的第一步。 .config 文件位于源代码根目录下, 该文件中包含所有内核配置选项的描述, 可以借助内核配置工具来选择这些选项。每个内核配置选项都有相关的名字和值。其名字的形式为 CONFIG_<NAME>, 其中 <NAME> 是对相关选项的标识; 变量可以有三个值: y、m 或 n。y 代表 “yes”, 表示该选项将会被编译到内核源代码中, 或者说会被构建到系统中。m 代表 “module”, 表示该选项将会以独立于内核源代码的模块的方式编译到内核中。如果未选择该选项 (即将该选项的变量值设为 n, 代表 “no”), 那么 .config 文件中就会出现下列注释: CONFIG_<NAME> is not set。 .config 文件中的选项的位置根据它们在内核配置工具中的顺序进行排序, 注释部分说明该选项位于哪个菜单下。我们来看看一个 .config 文件的节选:

```
-----
.config
1  #
2  # Automatically generated make config: don't edit
3  #
4  CONFIG_X86=y
5  CONFIG_MMU=y
6  CONFIG_UID16=y
7  CONFIG_GENERIC_ISA_DMA=y
8
9  #
10 # Code maturity level options
11 #
12 CONFIG_EXPERIMENTAL=y
13 CONFIG_CLEAN_COMPILE=
14 CONFIG_STANDALONE=y
15 CONFIG_BROKEN_ON_SMP=y
16
17 #
18 # General setup
19 #
20 CONFIG_SWAP=y
21 CONFIG_SYSVIPC=y
22 #CONFIG_POSIX_MQUEUE is not set
23 CONFIG_BSD_PROCESS_ACCT=y
-----
```

上述 .config 文件指出第 4~7 行的选项位于顶层菜单中, 第 12~15 行的选项位于代码成熟度选项菜单中, 第 20~23 行的选项位于通用设置选项菜单中。

所有配置工具都会产生上述菜单, 并且已经看到代码成熟度菜单项及通用设置菜单项等选项都位于顶层。后面两个选项被扩展为包含多个选项的子菜单。这些菜单都是在调用 make xconfig

命令时，由 qconf 配置工具提供的。配置工具显示的菜单都默认用于 X86 体系结构。图 9-6 给出了 PPC 体系结构所使用的菜单，在 make xconfig 调用的末尾，须追加参数 ARCH=ppc。

调用 make bzImage 构建内核映像时，顶层 makefile 文件会读取由配置工具产生的.config 文件。顶层 makefile 文件中包含了特定体系结构的 Makefile 文件提供的信息，它位于 arch/<arch>/目录下，该 Makefile 文件中包含了与平台相关的信息，通过 include 指令来实现。

```
-----
Makefile
434 include .config
...
450 include $(srctree)/arch/$(ARCH)/Makefile
-----
```

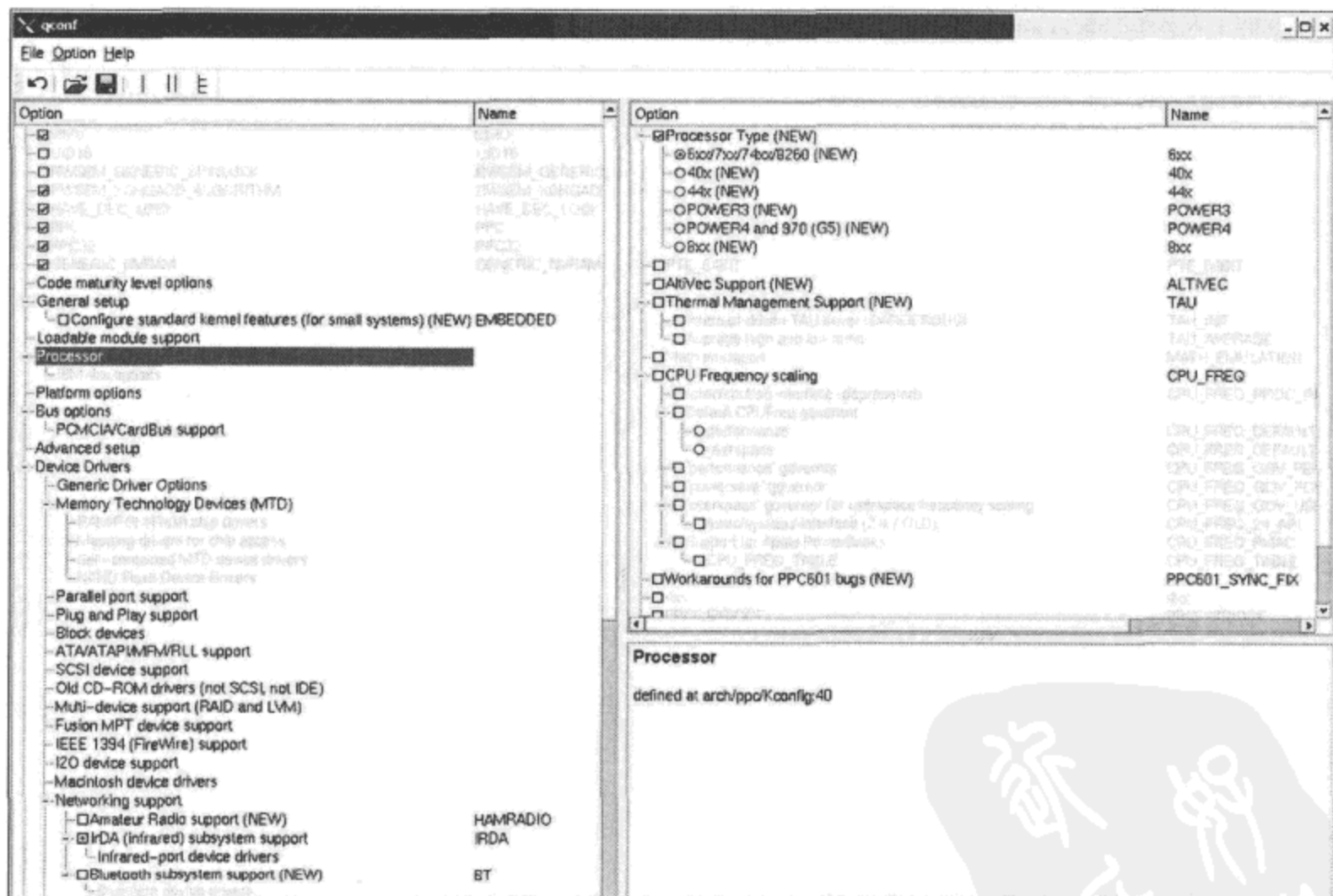


图 9-6 qconf 快照

由此可见，Makefile 文件已经确定编译的目标体系结构了。顶层 Makefile 文件通过以下三种可能的方式来确定体系结构：

- (1) 借助命令行参数 ARCH；
- (2) 借助环境变量 ARCH；
- (3) 在执行构建的宿主机上调用 uname，并自动接收由此命令产生的信息。

如果编译时所面向的体系结构与执行编译过程的宿主机体系结构不同时，就必须传递参数 CROSS_COMPILE，这是一个前缀，表示使用交叉编译器，或者，通过自行编辑 Makefile，并给该

变量赋值。例如，当在 x86 宿主机上为 PPC 处理器编译代码时，将执行下面的命令：

```
lkp:~#make xconfig ARCH=ppc
lkp:~#make ARCH=ppc CROSS_COMPILE=ppc-linux-
```

.config 文件也会产生 include/linux/autoconf.h 文件，#define 宏中包含已选择的 CONFIG_<NAME>值，而#undef 宏包含未选择的 CONFIG_<NAME>值。

2. 子makefile文件

构建一个系统依赖于各个子目录下的子 Makefile 文件。每个子目录下的 Makefile(被称为 sub-Makefile 或 kbuild Makefile) 都定义了根据该子目录下的源码文件编译目标文件的规则，并且仅对该目录下的文件作适当的修改。我们采用递归的方式调用位于 init/、drivers/、sound/、net/、lib/ 和 usr/等目录下的各个子目录中的子 Makefile 文件。

在递归调用之前，kbuild 首先要确定是否已经满足一些必要的条件，包括在必要时更新 include/linux/version.h 文件，并设置符号链接 include/asm，使之指向我们所编译的目标体系结构特有的文件。例如，如果为 PPC 编译代码，则 include/asm 指向 include/asm-ppc。kbuild 还要对文件 include/linux/autoconf.h 和 include/linux/config 进行编译。之后，从根目录开始进行递归。

如果你是一个内核开发者，当你向某一特殊的子系统添加了自己的代码时，要将你的文件放在一个特定的子目录中，并在必要时更新 makefile 文件以包括一些修改。如果你的代码是添加在一个已有的文件中，就可以使用 #ifdef(CONFIG_<NAME>) 块来把你的代码包围起来。如果在 .config 文件中选中这个值，那么在 include/linux/autoconf.h 文件中就包含相应的 #define 语句，就说明你的宏是有效的，而且在编译时会把你的修改包含进去。

我们必须遵守子 Makefile 文件特有的格式，它指出该如何编译目标文件。这些子 Makefile 文件都很简单，因为诸如编译程序名、要使用的函数库等都已经顶层 Makefile 文件以及与特定于体系结构的顶层 Makefile 文件中定义了，要遵循的编译规则在 scripts/Makefile.*s 文件中定义。子 Makefile 文件可能编译的文件列表如下。

□ \$(obj-y)：将被链接到 built-in.o，最终编译到 vmlinux 的目标文件列表。

□ \$(obj-m)：编译成模块的目标文件列表。

□ \$(lib-y)：编译到 lib.a 的目标文件列表。

换句话说，当我们执行 make bzImage 类型的调用时，kbuild 构建所有属于 obj-y 类型的目标文件并将它们链接到一起。所以子 Makefile 文件的要素之一就是指明类型。

```
obj-$(CONFIG_FOO) += foo.o
```

如果顶层 Makefile 文件读入的 .config 文件中的 CONFIG_FOO 被设置成 y，那么这一行就等价于 obj-y += foo.o。随后，kbuild 根据 scripts/Makefile.build 文件中定义的规则将该目录下的 foo.c 或 foo.S 文件编译为目标文件（稍后还有 scripts/Makefile.build 文件的更多介绍）。若 foo.c 或 foo.S 文件不存在，则显示如下语句：

```
Make[1]: *** No rule to make target '<subdir>/foo.o', needed by
'<subdir>/built-in.o'. Stop.
```

kbuild 通过对 obj-y 或 obj-m 显式地附加条件进行递归。你可以添加一个目录来设置 obj-y, 表明 kbuild 将会访问这个指定目录:

```
Obj-$(CONFIG_FOO) += /foo
```

若/foo 不存在, 则显示如下语句:

```
Make[2]: *** No rule to make target '<dir>/foo/Makefile'. Stop.
```

CML2

选择内核选项时, 你有没有想过我们所使用的配置工具在何处获得相关信息? kbuild 是以 CML2 为基础的。CML2 是一种专门用于内核配置的特定领域的语言, 它产生规则库 (rulebase), 解释器将读取并利用这些规则产生 config 文件, 该 config 文件包含 CML2 语言的语法规则和语义。配置程序读取的 CML2 规则库存放在 defconfig 和 Kconfig 文件中。defconfig 文件位于特定体系结构的顶层目录 arch/*/中, Kconfig 文件在其他大多数子目录中都可以找到, 该文件包含了创建的选项信息, 例如应该在该文件中列出的菜单、应提供的帮助信息、config 名称值, 以及只编译到内核中还是同时编译成模块的选择信息。有关 CML2 和 Kconfig 文件的更多信息, 详情可参阅文件 Documentation/kbuild/kconfig-language.txt。

让我们回顾一下有关 kbuild 操作的执行过程, 第一步是通过命令 make xconfig 或 make xconfig ARCH=ppc 调用配置工具进行配置, 使用哪个命令取决于构建过程所面向的体系结构。我们对该工具进行的选择会被保存在.config 文件中。发出一个诸如 make bzImage 的内核构建调用时, 顶层的 Makefile 文件会读取.config 的内容。顶层 Makefile 文件向下一级目录递归之前, 将执行以下操作:

- (1) 更新 include/linux/version.h.
- (2) 设置符号链接 include/asm, 使之指向编译过程所面向的特定目标体系结构的文件。
- (3) 构建 include/linux/autoconf.h.
- (4) 构建 include/linux/config.h.

随后 kbuild 操作会递归到下一级目录中, 对所有的子 Makefile 文件进行 make 操作, 并在每个子目录中创建目标文件。

我们已经讨论了子 Makefile 文件的结构, 现在我们来研究一下顶层 Makefile 文件, 看看驱动内核编译时如何使用它。

3. Linux内核的Makefile文件

Linux 的 Makefile 文件相当复杂。本节旨在说明源码树下所有子 Makefiles 文件之间的关系, 以及在这些文件中执行 make 命令的细节。但是, 如果你想通过理解 kbuild Makefile 文件的所有细节, 来拓展有关 make 知识, 是不错的起点。有关 make 的更多信息, 可以登录 www.gnu.org/software/make/make.html。

事实上, 源码树的几乎每个子目录下都有一个 makefile 文件。如前所述, 子树的 Makefile 主要负责特殊种类的源码 (或内核子系统), 这些文件相当简单, 并仅定义了要加入源码表的目标源文件, 稍后再编译这些文件。同时, 另有 5 个其他 Makefile 文件定义规则并执行这些文件, 它

们就是顶层 makefile 文件：arch/\$(ARCH)/Makefile、scripts/Makefile.build、scripts/Makefile.clean 和 scripts/Makefile。图 9-7 给出了各个 Makefile 文件间的关系。可定义的关系类型有“include”和“execute”。提及“include”关系类型时，是指 Makefile 要通过 include <filename> 规则添加其他文件的信息。提及“execute”关系类型时，是指上一级 Makefile 通过 make-f 命令调用它的下一级 makefile 文件。

在源码树的根目录下发出一个 make 调用时，就会调用顶层 Makefile 文件。顶层 Makefile 文件定义了随后要输出到其他 Makefile 的变量，还进一步对顶层源代码子目录中的每个 Makefile 发出 make 调用，终止对它们执行的操作。

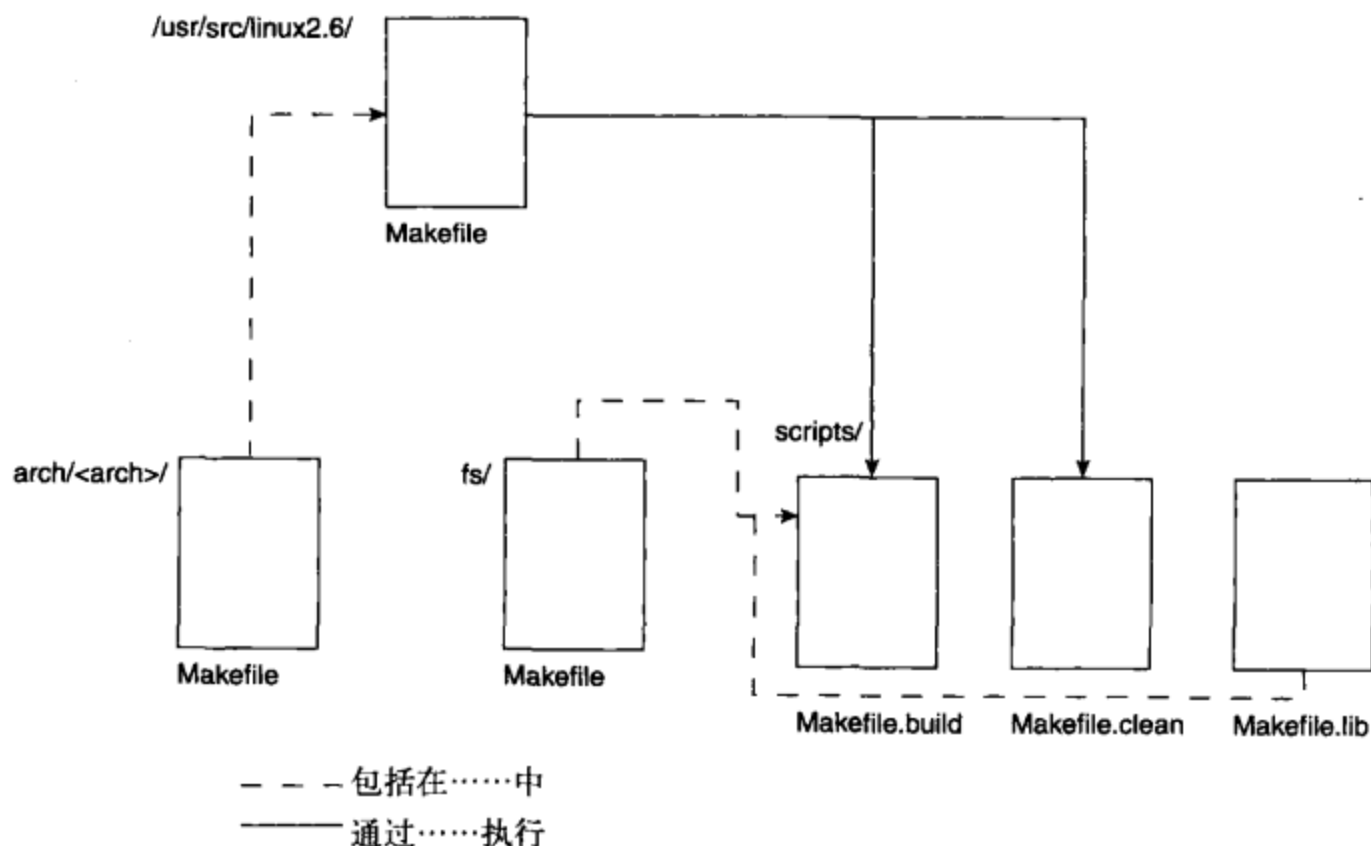


图 9-7

编译程序和链接程序的调用在 scripts/Makefile.build 中定义。就是说，向下一级子目录递归并通过 make 命令编译目标文件时，其实是执行了定义在文件 Makefile.build 中的一些规则。这可以借助 \$(Q) \$(MAKE) \$(build)=<dir> 的简写形式的调用来实现，这条规则就是 make 在每个子目录中得以调用的方式。build 变量被简写为：

```
Makefile
1157 build := -f $(if $(KBUILD_SRC),$(srctree)/)scripts/Makefile.build obj
```

调用 \$(Q) \$(MAKE) \$(build)=fs 可扩展为：

```
"@ make -f /path/to/source/scripts/Makefile.build obj=fs".
```

随后，scripts/Makefile.build 读取作为参数（本例中是 fs）传递的目录中的 Makefile 文件。子 Makefile 文件中定义了一个或多个 obj-y、obj-m、lib-y 等列表。之后，文件 scripts/Makefile.build 会根据 scripts/Makefile.lib 文件中的定义，对这些子目录中的源文件进行编译，然

后再进一步递归到列表中定义的任意子目录，调用的方式与前面相同。

举例说明，若我们可以通过内核配置工具得到文件系统菜单，并选择了 Ext3 日志文件系统，那么.config 文件中的 CONFIG_EXT3_FS 就会被设置为 y，下面是子 Makefile 文件中相应 fs 的片断：

```
-----
Makefile
49  obj-$(CONFIG_EXT3_FS) += ext3/
-----
```

当 make 使用这种规则运行时，会进行 obj-y += ext3/ 操作，即把 ext3/ 作为 obj-y 中的一项。make 自动辨识出它是一个子目录，并调用 \$(Q) \$(MAKE) \$(build)=ext3。

\$(Q)

在所有 \$(MAKE) 调用前都要加一个 \$(Q) 变量。对于 2.6 内核树及 kbuild 基础结构的整理，你可以禁止 make 冗长的输出格式。make 在执行之前先输出命令行。当某一行以 @ 开头时，那么表示不必输出这一行：

```
-----
Makefile
254  ifeq ($(KBUILD_VERBOSE),1)
255    quiet =
256    Q =
257  else
258    quiet=quiet_
259    Q = @
260  endif
-----
```

从上述几行可以看出，如果 KBUILD_VERBOSE 被设置为 0 则 Q 被设置为 @，目的是简化编译语句。

当构建过程结束后，就会产生一个内核映像文件。因为内核是通过 zlib 算法进行压缩的，故可启动的压缩内核映像文件被称作 zImage 或 vmlinuz。常见的 Linux 规范也指定可引导映像 在文件系统中的位置，一般必须位于 /boot 或 / 目录下。到此，内核映像文件就可以通过引导加载 程序装载到内存中。

9.3 小结

本章探究目标文件的编译、链接过程以及目标文件的结构，以便理解可执行代码的最终形式。同时我们考察了围绕内核编译系统的一些基本结架，以及源代码的结构如何与构建系统本身相结合。由于前面章节已有相关讨论，本章只作简要介绍。

9.4 习题

1. 描述不同类型的 ELF 文件，并说明它们的作用。

2. 目标文件中的段是指什么?
3. 考察文件 `arch/ppc/Kconfig` 及 `arch/i386/Kconfig`, 给出每种体系结构所支持的处理器类型。
4. 考察文件 `arch/ppc` 及 `arch/i386`, 它们所共有的文件和目录有哪些? 研究这些文件并列出它们提供的支持, 看看它们是否完全匹配?
5. 如果你要交叉编译内核, 那么你给交叉编译程序指定前缀时所使用的参数是什么?
6. 在什么情况下, 你要通过命令行参数 `ARCH` 来指定系统的体系结构?
7. 什么是子 `makefile` 文件? 它们如何工作?
8. 考察文件 `scripts/Makefile.build`, `scripts/Makefile.clean` 以及 `scripts/Makefile.lib`, 分别列举其功能。



第 10 章

向内核添加代码

本章内容

- 浏览源代码
- 编写代码
- 构建和调试

本章分为“浏览源代码”和“编写源代码”两大部分。

10.1 节描述几乎所有 Linux 系统都通用的一个设备驱动程序，它存放于目录/dev/random 下，并说明内核如何与之通信。同时，再次像以前一样描述内核的内部工作，并展示其实用之处。

10.2 节将构建一个设备驱动程序，并深入分析编写设备驱动程序时会遇到的常见问题。

此后将讲述如何使用/proc 系统来调试设备驱动程序。也许这就是一个硬币的第三面？

10.1 浏览源代码

本节将介绍 Linux 下的系统调用及设备驱动程序（也称为模块）的概念。用户程序通过系统调用向操作系统请求服务，因而可以通过添加系统调用来创建新的内核服务。第 3 章描述了系统调用的内部实现。本章将介绍如何将自定义系统调用合并到 Linux 内核中，从而展示其实用的一面。

设备驱动程序涉及 Linux 内核利用的接口，让开发者可以控制系统的输入/输出设备。全书都是围绕 Linux 设备驱动程序来撰写的，而本章则是这一主题的精华所在。本节沿着这样一个思路来介绍驱动程序，首先说明在文件系统中如何表示设备，然后分析控制设备的具体内核代码。下一节将介绍如何利用所学知识构建一个实用的字符驱动程序。本章的最后将介绍如何编写系统调用，以及如何构建内核。我们首先来研究一下文件系统，然后说明文件如何与内核相关联。

10.1.1 熟悉文件系统

通过/dev 可以访问 Linux 的设备。例如，执行命令 `ls -l /dev/random`，生成的结果如下：

```
crw-rw-rw- 1 root root 1, 8 Oct 2 08:08 /dev/random
```

首字母 c 表示该设备是一个字符设备，若为 b 则表示块设备。接下来的两列是用户及用户所在的组，后面是以逗号隔开的是两个数字（这里是 1 和 8），第一个数字是设备的主设备号，第

二个是从设备号。设备驱动程序向内核注册时就注册了其主设备号。打开一个给定的设备时，内核将通过设备文件的主设备号^①找到以主设备号注册的设备驱动程序。从设备号通过内核传递给设备驱动程序本身，这是因为一个单独的驱动程序可以控制多个设备。例如，`/dev/urandom` 设备的主设备号为 1 而从设备号为 9，表示以主设备号为 1 注册的设备驱动程序可以处理设备 `/dev/random` 和 `/dev/urandom`。

要产生一个随机数，只需读取 `/dev/random`。下面是读取 4 字节随机数^②的一种方法：

```
lkp@lkp:~$ head -c4 /dev/urandom | od -x
0000000 823a 3be5
0000004
```

若重复执行这条命令，会发现产生的 4 字节随机数[823a 3be5]总在变化。为了举例说明 Linux 内核如何使用设备驱动程序，我们来分析用户访问 `/dev/random` 时内核的操作步骤。

我们已经知道 `/dev/random` 设备文件的主设备号是 1，并可以通过查看 `/proc/devices` 文件来确定驱动程序控制哪个设备节点。

```
lkp@lkp:~$ less /proc/devices
Character devices:
1 mem
```

让我们以 `mem` 设备驱动程序为例来看看“随机数”是如何产生的。

```
-----
drivers/char/mem.c
653 static int memory_open(struct inode * inode, struct file * filp)
654 {
655     switch (imajor(inode)) {
656         case 1:
657             ...
676         case 8:
677             filp->f_op = &random_fops;
678             break;
679         case 9:
680             filp->f_op = &urandom_fops;
681             break;
-----
```

第 655~681 行：switch 条件语句根据操纵的设备的从设备号来初始化驱动程序的数据结构，尤其是设置 `filp` 和 `fops`。

我们不禁想问：“什么是 `flip`? 什么是 `fop`?”

10.1.2 `flip` 和 `fops`

`flip` 只是一个文件结构指针，而 `fop` 是一个 `file_operations` 结构指针。内核通过 `file_operations` 结构来确定操作文件时要调用的函数。下面是 `file_operations` 结构中用于随机设备驱动程序的部分内容。

① 使用命令 `mknod` 可创建块设备文件和字符设备文件。

② `head-c4` 用于聚集前 4 字节，`od-x` 以十六进制来表示获得的随机数。

```

-----
include/linux/fs.h
556 struct file {
557     struct list_head f_list;
558     struct dentry *f_dentry;
559     struct vfsmount *f_vfsmnt;
560     struct file_operations *f_op;
561     atomic_t f_count;
562     unsigned int f_flags;
...
581     struct address_space *f_mapping;
582 };
-----

include/linux/fs.h
863 struct file_operations {
864     struct module *owner;
865     loff_t (*llseek) (struct file *, loff_t, int);
866     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
867     ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
868     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
869     ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
870     int (*readdir) (struct file *, void *, filldir_t);
871     unsigned int (*poll) (struct file *, struct poll_table_struct *);
872     int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
...
888 };
-----

```

随机设备驱动程序以如下方式声明它所提供的文件操作：驱动程序所实现的函数必须符合 `file_operations` 结构中所列出的函数原型：

```

-----
drivers/char/random.c
1824 struct file_operations random_fops = {
1825     .read    = random_read,
1826     .write   = random_write,
1827     .poll    = random_poll,
1828     .ioctl   = random_ioctl,
1829 };
1830
1831 struct file_operations urandom_fops = {
1832     .read    = urandom_read,
1833     .write   = random_write,
1834     .ioctl   = random_ioctl,
1835 };
-----

```

第 1824~1829 行：随机设备提供的操作有 `read`、`write`、`poll` 以及 `ioctl`。

第 1831~1835 行：`urandom` 设备提供的操作有 `read`、`write` 以及 `ioctl`。

`poll` 操作允许程序员在执行某种操作之前查看该操作是否阻塞。它指出当请求的熵的字节

多于熵池^①中的字节时，`/dev/random` 就会阻塞，事实也的确如此。但 `/dev/urandom` 设备不会被阻塞，只是当熵池太小时，它可能无法返回完整的随机数。详情可查看系统的帮助页面，具体来说，是使用命令 `man 4 random`。

进一步研究代码时发现，对 `/dev/random` 进行读操作时，内核将控制权交给函数 `random_read()` (参见代码第 1825 行)。`random_read()` 定义如下：

```
-----
drivers/char/random.c
1588 static ssize_t
1589 random_read(struct file * file, char __user * buf, size_t
nbytes, loff_t * ppos)
-----
```

该函数的参数如下。

- **file**: 指向设备文件的结构体。
- **buf**: 指向存放结果的用户空间。
- **nbytes**: 请求数据的大小。
- **ppos**: 指向文件中用户访问的位置。

也许你会想到这样一个问题：如果设备驱动程序在内核空间中执行，但缓冲区却位于用户空间，那我们如何才能安全访问 `buf` 中的数据？下一节将解释数据在用户空间和内核空间之间移动的过程。

10.1.3 用户空间和内核空间

若想仅通过 `memcpy()` 便把数据从内核空间复制到用户空间，未必行得通，因为执行 `memcpy()` 操作时用户空间的地址可能已被交换出去。Linux 提供的函数 `copy_to_user()` 及 `copy_from_user()` 使得驱动程序可以在内核空间和用户空间中传递数据。在函数 `read_random()` 中，通过 `extract_entropy()` 函数来实现这一功能，但还有一些曲折：

```
-----
drivers/char/random.c
1: static ssize_t extract_entropy(struct entropy_store *r, void * buf,
2:      size_t nbytes, int flags)
3: {
1349 static ssize_t extract_entropy(struct entropy_store *r, void * buf,
1350      size_t nbytes, int flags)
1351 {
...
1452      /* Copy data to destination buffer */
1453      i = min(nbytes, HASH_BUFFER_SIZE*sizeof(__u32)/2);
1454      if (flags & EXTRACT_ENTROPY_USER) {
1455          i -= copy_to_user(buf, (__u8 const *)tmp, i);
1456          if (!i) {
1457              ret = -EFAULT;

```

① 在随机设备驱动程序中，熵指不可预测的系统数据，常见的熵的来源有：键盘输入，鼠标移动以及其他无规律的输入等。

```

1458         break;
1459     }
1460 } else
1461     memcpy(buf, (__u8 const *)tmp, i);
-----

```

`extract_entropy()` 包含如下参数：

- **r** 是指向熵内部存储器的指针，为了便于讨论我们不予考虑；
- **buf** 是指向内存区的指针，该内存区应当已填充数据；
- **nbytes** 是写入 **buf** 的数据大小；
- **flags** 告诉函数 **buf** 是在内核空间还是用户空间；
- **extract_entropy()** 将返回 `ssize_t`，表明已产生的随机数占多少字节。

第 1454~1455 行：若 **flags** 表明指针 **buf** 指向用户空间中的某个位置，那么我们将使用函数 `copy_to_user()` 将内核空间的内容复制给用户空间，其中指针 **tmp** 和 **buf** 分别指向内核空间和用户空间。

第 1460~1461 行：若指针 **buf** 指向内核空间的某一位置，则可使用函数 `memcpy()` 来复制数据。

获得随机字节是内核空间以及用户空间程序都可能涉及的事情；内核空间的程序可以通过不设置标志位来避免函数 `copy_to_user()` 带来的额外开销。例如，内核实现一个加密的文件系统，可以避免把数据复制到用户空间的额外开销。

10.1.4 等待队列

我们先不考虑数据如何在内核空间和用户空间之中传输，而是回到 `read_random()` 函数，看看它如何使用等待队列。

有时，驱动程序可能需要等待某些条件的成立，也许要访问系统资源。此时，我们并不希望内核一味等待访问的结束。当等待发生时，因为所有系统处理都得暂停而引起内核等待就会出现^①。声明一个等待队列，可以将等待某种条件的驱动程序延迟到条件满足时再进行处理。

正在等待的进程使用了两个数据结构，即等待队列和等待队列头。模块应当创建等待队列头，并利用使用宏 `sleep_on`、`wake_up` 的模块部分进行管理。函数 `random_read()` 恰是如此：

```

-----
drivers/char/random.c
1588 static ssize_t
1589 random_read(struct file * file, char __user * buf, size_t nbytes, loff_t
*ppos)
1590 {
1591     DECLARE_WAITQUEUE(wait, current);
...
1597     while (nbytes > 0) {
...
1608         n = extract_entropy(sec_random_state, buf, n,
1609                             EXTRACT_ENTROPY_USER |

```

^① 实际上，运行内核任务的 CPU 会暂停运行。对于多处理器系统来说，其他 CPU 尚可继续运行。


```

1610         EXTRACT_ENTROPY_LIMIT |
1611         EXTRACT_ENTROPY_SECONDARY);
...
1618     if (n == 0) {
1619         if (file->f_flags & O_NONBLOCK) {
1620             retval = -EAGAIN;
1621             break;
1622         }
1623         if (signal_pending(current)) {
1624             retval = -ERESTARTSYS;
1625             break;
1626         }
...
1632         set_current_state(TASK_INTERRUPTIBLE);
1633         add_wait_queue(&random_read_wait, &wait);
1634
1635         if (sec_random_state->entropy_count / 8 == 0)
1636             schedule();
1637
1638         set_current_state(TASK_RUNNING);
1639         remove_wait_queue(&random_read_wait, &wait);
...
1645         continue;
1646     }

```

第 1591 行：初始化当前任务的等待队列 `wait`。宏 `current` 是一个指针，指向当前任务的 `task_struct` 结构。

第 1608~1611 行：从设备获取一组随机数据。

第 1618~1626 行：若无法从熵池中获取所需数目的熵，并且未被阻塞或有挂起的信号时，给调用者返回错误信息。

第 1631~1633 行：建立等待队列。`random_read()` 使用自己的等待队列 `random_read_wait`，而不是系统的等待队列。

第 1635~1636 行：此时，读操作被阻塞，若无用于读取的熵，则需调用函数 `schedule()` 来释放处理器的控制权。（变量 `entropy_count` 包含位信息而不是字节信息，因而可以通过除以 8 来判断熵池是否已满。）

第 1638~1639 行：当我们终于重新开始时，清除等待队列。

注意 Linux 中的 `random` 设备要求当熵池队列满时才返回一个随机数，但 `urandom` 设备并无这种要求，它返回时并不考虑熵池中可用数据的大小。

让我们进一步看看当某个任务调用函数 `schedule()` 时会执行哪些相关操作：

```

-----
kernel/sched.c
2184 asmlinkage void __sched schedule(void)
2185 {
...
2209     prev = current;

```

```

...
2233  switch_count = &prev->nivcsw;
2234  if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
2235      switch_count = &prev->nvcsw;
2236      if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
2237          unlikely(signal_pending(prev))))
2238          prev->state = TASK_RUNNING;
2239      else
2240          deactivate_task(prev, rq);
2241  }
2242  ...

```

第 2209 行：变量 `prev` 中存储的是一个指针，指向当前任务的任务结构。当该任务自己调用函数 `schedule()` 时，指针 `current` 就会指向它的结构体。

第 2233 行：我们把任务的上下文切换计数器 `nivcsw` 存放在 `switch_count` 中，切换成功后 `nivcsw` 就会加 1^①。

第 2234 行：只有当任务的状态变量 `prev->state` 非零且不存在内核抢占时，才执行该条件语句。换句话说，当任务的状态不是 `TASK_RUNNING` 且内核并未抢占该任务时，才执行该语句。

第 2235~2241 行：如果该任务是可中断的，那么它肯定会交出处理器的控制权。若应该释放控制权的任务还有挂起的信号，则将其状态设置为 `TASK_RUNNING`，以便向其他任务传递控制权之后，这个任务仍然有机会再次被调度程序选中投入运行。若无挂起的信号（这是普遍情况），则使该任务处于非活动状态，并将 `switch_count` 赋给 `nvcsw`。随后，调度程序将 `switch_count` 的值加 1，因此 `nvcsw` 或 `nivcsw` 也进行了加 1 操作。

然后，函数 `schedule()` 在调度程序的运行队列中选出另一个任务，并把控制权交给它^②。

如果当前任务由于某种原因需要等待，可以调用函数 `schedule()` 将处理器的控制权交给另一个内核任务。内核中的其他任务就利用这个机会获得处理器来完成自身的工作，当控制权交还给调用 `schedule()` 函数的任务时，导致该任务等待的原因已被移除。

再回到函数 `random_read()` 上来。最后，内核把控制权又交给了 `random_read()` 函数，然后清空等待队列，并继续执行。这将重复前面的操作，如果系统中已产生足够的熵，我们就能返回所请求的随机字节数。

`random_read()` 函数在调用函数 `schedule()` 前将自己的状态设置为 `TASK_INTERRUPTIBLE`，以便处于等待队列时可以通过信号来唤醒。调用函数 `batch_entropy_process()` 和 `random_ioctl()` 中的 `wake_up_interruptible()`，可以收集额外的熵，此时驱动程序自身的代码就会产生这些可唤醒自己的信号。通常，若该任务等待硬件的响应而不是软件的响应，其状态被设置为 `TASK_UNINTERRUPTIBLE`（反之则被设置为 `TASK_INTERRUPTIBLE`）。

函数 `random_read()` 用于将控制权传给其他任务，实现这部分功能的代码（参见 `drivers/char/random.c` 中的第 1632~1639 行）源自调度程序代码中函数 `interruptible_sleep_on()` 的变体。

① 如何使用上下文切换计数器可参阅第 4 章和第 7 章。

② 如要了解详细信息，可参阅第 7 章中的“`switch_to()`”部分。

```

-----
kernel/sched.c
2489 #define SLEEP_ON_VAR          \
2490     unsigned long flags;        \
2491     wait_queue_t wait;          \
2492     init_waitqueue_entry(&wait, current);
2493
2494 #define SLEEP_ON_HEAD          \
2495     spin_lock_irqsave(&q->lock, flags); \
2496     __add_wait_queue(q, &wait); \
2497     spin_unlock(&q->lock);
2498
2499 #define SLEEP_ON_TAIL          \
2500     spin_lock_irq(&q->lock); \
2501     __remove_wait_queue(q, &wait); \
2502     spin_unlock_irqrestore(&q->lock, flags);
2503
2504 void fastcall __sched interruptible_sleep_on(wait_queue_head_t *q)
2505 {
2506     SLEEP_ON_VAR
2507
2508     current->state = TASK_INTERRUPTIBLE;
2509
2510     SLEEP_ON_HEAD
2511     schedule();
2512     SLEEP_ON_TAIL
2513 }
-----

```

q 是一个 wait_queue_head 型数据结构，用于调整模块的休眠和等待状态。

第 2494~2497 行：自动将任务添加到等待队列 q 中。

第 2499~2502 行：自动从等待队列 q 中删除任务。

第 2504~2513 行：添加到等待队列，并将处理器的控制权交给另一任务。当获得对处理器的控制权后，从队列中删除自身。

函数 random_read() 使用自己的等待队列代码而不是标准宏，但本质上还是使用 interruptible_sleep_on() 函数来处理异常，只要 random_read() 函数还有多个可用的熵，那它就不会让出处理器，而是再次循环以便获取所有请求的熵。但若没有足够的熵，random_read() 就会处于等待状态，直到驱动程序获取了可使用的熵之后，由 wake_up_interruptible() 函数将它唤醒。

10.1.5 工作队列及中断

Linux 中的设备驱动程序都须例行公事地处理由与之交互的设备产生的中断。中断可以触发设备驱动程序中的中断处理程序，并暂停当前用户空间和内核空间正在执行的代码。显然，驱动程序中的中断处理程序执行得越快越好，以免长时间占用内核。

尽管如此，我们还是遇到了中断处理中标准的难题：如何处理一项任务繁重的中断呢？通常的做法是使用上半部例程和下半部例程。上半部例程可以快速接收中断并调度下半部例程，而由下半部例程来完成主要工作，下半部分一有机会就被执行。通常，上半部例程在执行时禁用中断，以确保一个中断处理程序不会被同一中断再次中断，因而设备驱动程序不必处理递归中断。下半

部例程通常在启用中断的情况下运行，以便在它继续执行大部分工作时处理其他中断。

在 Linux 以前的版本中，任务队列划分的上半部分和下半部分也被称为快速中断和慢速中断。Linux 2.6 内核中引入了工作队列的概念，现在工作队列已是处理下半部中断的标准途径。

当内核接收到中断时，处理器应当停止当前任务的运行，以便立即处理中断。当 CPU 进入这一模式时，通常就说它处于中断上下文中。此时，内核确定要将控制权交给哪个中断处理程序。当设备驱动程序想处理中断时，它将使用函数 `request_irq()` 来请求该中断号，并在该中断发生时注册被调用的中断处理函数，该操作通常在模块初始化时进行。由 `request_irq()` 注册的上半部中断处理函数只完成最基本的管理工作，然后将余下的部分以任务的形式插入工作队列，以便等候处理。

和上半部机制中使用的 `request_irq()` 函数一样，工作队列通常也是在模块初始化时注册的。它们可以通过宏 `DECLARE_WORK()` 静态初始化，也可以调用 `INIT_WORK()` 来分配任务结构体，并进行动态初始化。下面是对这些宏的定义：

```
-----
include/linux/workqueue.h
30 #define DECLARE_WORK(n, f, d) \
31     struct work_struct n = __WORK_INITIALIZER(n, f, d)
...
45 #define INIT_WORK(_work, _func, _data) \
46     do { \
47         INIT_LIST_HEAD(&(_work)->entry); \
48         (_work)->pending = 0; \
49         PREPARE_WORK((_work), (_func), (_data)); \
50         init_timer(&(_work)->timer); \
51     } while (0)
-----
```

上述两种宏均使用了如下参数。

- **n** 或者 **work**: 创建或初始化的任务结构体名称。
- **f** 或者 **func**: 当从工作队列中删除一个任务结构体时所执行的函数。
- **d** 或者 **data**: 用于保存函数 **f** 或 **func** 运行时传递给它们的数据。

在函数 `register_irq()` 中注册的中断处理函数用于接收中断，并将相关数据从中断处理程序的上半部分传递给其下半部分，这是通过设置数据结构 `work_struct` 并调用工作队列中的 `schedule_work()` 函数来实现的。

工作队列函数中的代码可在进程上下文中执行，并可完成在中断上下文中不可能做的事情，例如向用户空间写入数据，复制用户空间的数据以及睡眠等。

Tasklet 和工作队列相似，但完全在中断上下文中运行。当下半部的工作量很少时，Tasklet 大有用处，它能够节省中断处理函数中上半部和下半部的开销。宏 `DECLARE_TASKLET()` 可以初始化 Tasklet。

```
-----
include/linux/interrupt.h
136 #define DECLARE_TASKLET(name, func, data) \
137 struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }
-----
```

- **name**: 创建的 tasklet 结构体的名称。
- **func**: 调度 tasklet 时需要调用的函数。
- **data**: 用于保存执行 tasklet 时传给函数 func 的数据。

要调度 tasklet, 需要使用函数 `tasklet_schedule()`:

```
-----
include/linux/interrupt.h
171 extern void FASTCALL(__tasklet_schedule(struct tasklet_struct *t));
172
173 static inline void tasklet_schedule(struct tasklet_struct *t)
174 {
175     if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
176         __tasklet_schedule(t);
177 }
-----
```

- **tasklet_struct**: 通过 `DECLARE_TASKLET()` 创建的 tasklet 的名称。

在上半部中断处理程序中, 可以调用函数 `tasklet_schedule()`, 以确保将来某一时刻, 在 tasklet 中声明的函数一定会被执行。tasklet 与工作队列的不同之处在于, 不同的 tasklet 可以同时不同的 CPU 上执行。若已调度的 tasklet 在执行前被再次被调度, 那么该 tasklet 仅执行一次。当 tasklet 在中断上下文中运行时, 它不能休眠也不能向用户空间复制数据。因为 tasklet 运行在中断上下文中, 如果不同的 tasklet 需要通信, 唯一安全的同步方式是使用自旋锁。

10.1.6 系统调用

除了通过设备驱动程序向内核添加代码外, 还有别的方式。在 Linux 内核系统中, 用户空间的程序可以通过系统调用来访问内核服务程序和系统硬件。用户态程序可用的许多 C 库例程都把一些代码以及一个或多个系统调用打包在一起, 以共同实现一个函数的功能。事实上, 通过内核代码也可以访问系统调用。

系统调用天生就与硬件相关。在 Intel 体系结构中, 所有系统调用都使用软中断 0x80。系统调用的参数都通过通用寄存器来传递, 在 x86 体系结构中, 系统调用最多可有 5 个参数。若需要 5 个以上的参数, 就只能传递一个指向参数块的指针。执行汇编指令 `int 0x80` 时, 处理器借助其异常处理能力来调用一个特定的内核态例程。

10.1.7 其他类型的驱动程序

迄今为止, 我们讨论的都是字符设备驱动程序, 它常常是最容易理解的, 你可能想编写一些以其他方式与内核打交道的驱动程序。

块设备与字符设备的相似之处在于, 通过文件系统就可以访问它们。系统中第一个 IDE 硬盘驱动器的设备文件是 `/dev/hda`。块设备注册和注销的方式与字符设备类似, 也是通过函数 `register_blkdev()` 和 `unregister_blkdev()` 来实现的。

块设备驱动程序与字符设备驱动程序的不同之处主要在于, 块设备驱动程序本身没有读写功能, 它们使用的是请求的方法。

2.6 版本的内核对块设备子系统做出了很大改动。以前的函数如 `block_read()` 和 `block_write()`，以及 `blk_size` 和 `blksize_size` 等内核数据结构都被删除了。本节仅着眼于 2.6 版本中块设备驱动程序的实现。

如果你的 Linux 内核必需支持磁盘设备（或类磁盘设备）。那么你就得编写一个块设备驱动程序，这个驱动程序必须告诉内核它将与哪种磁盘打交道。这就需要使用结构 `gendisk`：

```
-----
include/linux/genhd.h
82 struct gendisk {
83     int major;          /* major number of driver */
84     int first_minor;
85     int minors;
86     char disk_name[32]; /* name of major driver */
87     struct hd_struct **part; /* [indexed by minor] */
88     struct block_device_operations *fops;
89     struct request_queue *queue;
90     void *private_data;
91     sector_t capacity;
...
-----
```

第 83 行：`major` 是块设备的主设备号。可以静态设置它，也可以由 `register_blkdev()` 函数动态生成，这与字符设备完全一样。

第 84~85 行：`first_minor` 和 `minors` 用于确定块设备中的分区数，其中 `minors` 包含块设备中从设备号的最大值，`first_minor` 则是它的第一个从设备号。

第 86 行：`disk_name` 是一个 32 字符的块设备名，我们可以在文件系统 `/dev`、`sysfs` 和 `/proc/partitions` 中看到。

第 87 行：`hd_struct` 存放与该块设备相关的分区设置。

第 88 行：`fops` 是一个指向结构体 `block_operations` 的指针，该结构体包含操作 `open`、`release`、`ioctl`、`media`、`_changed` 以及 `revalidate_disk`（可参阅 `include/linux/fs.h` 文件）。在 2.6 版的内核中，每个设备都有自己的操作集。

第 89 行：指针 `request_queue` 指向一个队列，该队列可帮助管理设备中挂起的操作。

第 90 行：`private_data` 指向私有信息，内核的块设备子系统不能对其进行访问，其常见的应用是存放数据，这些数据用于低级的、与具体设备相关的操作。

第 91 行：`capacity` 描述以 512 字节扇区为单位的块设备大小。如果该设备是可移动的，如软盘和光盘，当 `capacity` 等于 0 时表示没有该设备。如果你的设备不是每扇区包含 512 字节，你仍然要这样设置。例如，如果你的设备使用了 1000 个 256 字节的扇区，那就相当于 500 个 512 字节的扇区。

除了需要 `gendisk` 数据结构外，块设备还需要使用自旋锁结构，以便与请求队列配合。

不论是自旋锁还是 `gendisk` 结构体中的字段都必须由设备驱动程序来初始化（登录 http://en.wikipedia.org/wiki/Ram_disk 可以获取有关初始化 RAM 磁盘块设备驱动程序的示例）。块设备初始化完毕并准备处理请求时，应当调用函数 `add_disk()` 将该设备添加到系统中。

最后，如果该块设备可以作为系统的熵源时，模块的初始化也可以调用函数 `add_disk_`

randomness() 来实现 (详情可查阅文件 drivers/char/random.c)。

既然已经了解块设备驱动程序初始化的基本知识, 那么我们就可以来看看它的全貌以及退出和清除该驱动程序等工作, 这在 Linux 2.6 中很容易实现。

del_gendisk (struct gendisk) 可将 gendisk 从系统中删除, 并清除其分区信息。但执行该调用之前必须先执行 putdisk (struct gendisk) 调用, 它用于释放内核中涉及 gendisk 的内容。可调用 unregister_blkdev(int major, char[16] device_name) 来注销块设备, 该函数允许我们释放 gendisk 结构。

我们也需要清除与块设备驱动程序相关的请求队列, 这可以使用系统调用 blk_cleanup_queue(struct*request_queue) 来实现。注意, 若只能通过 gendisk 来引用请求队列, 则应当在释放 gendisk 前调用 blk_cleanup_queue。

在对块设备的初始化和关闭进行概要说明时, 我们很容易不谈及有关请求队列的细节问题, 但既然已经建立了一个设备驱动程序, 那就必须实实在在地做点什么, 而请求队列就是完成块设备主要读写功能的部分。

```
-----
include/linux/blkdev.h
576 extern request_queue_t *blk_init_queue(request_fn_proc *, spinlock_t *);
...
-----
```

第 576 行: 为了创建请求队列, 需要使用 blk_init_queue, 并传给它一个指向自旋锁的指针和一个指向请求函数的指针, 其中自旋锁可控制对队列的访问, 而任何时候访问该设备都必须调用这个请求函数。该请求函数的原型如下:

```
static void my_request_function( request_queue_t *q );
```

通常, 任务繁重的请求函数可以很方便地使用许多辅助函数。为了确定下一个要处理的请求, 应当调用函数 elv_next_request(), 它将返回一个指向请求结构体的指针, 没有下一个请求时该函数返回 NULL。

在 Linux 2.6 内核中, 块设备驱动程序在请求结构体中遍历所有的 BIO 结构。BIO 代表块设备 I/O, include/linux/bio.h 中有它的完整定义。

BIO 结构包含一个指向 biovec 结构体链表的指针, 该结构体定义如下:

```
-----
include/linux/bio.h
47 struct bio_vec {
48     struct page *bv_page;
49     unsigned int bv_len;
50     unsigned int bv_offset;
51 };
-----
```

每个 biovec 都使用其页面结构来存放数据缓冲区, 缓冲区中的数据是从磁盘读出的或最终将写入磁盘。Linux 2.6 内核中有大量 bio 辅助例程, 可用于反复使用 bio 结构中的数据。

为了确定 BIO 操作的大小, 可以查看 BIO 结构中的 bio_size 字段, 得到的结果以字节为单

位,也可以使用 `bio_sectors()` 宏,得到以扇区来计算 BIO 操作的大小。块操作的类型,如 READ 或 WRITE,可由 `bio_data_dir()` 决定。

为了遍历 BIO 结构中的 `biovec` 链表,可以使用 `bio_for_each_segment()` 宏。在该循环中,可以使用更多的宏,以便深入研究 `biovec_bio_page()`、`bio_offset()`、`bio_curr_sectors()` 和 `bio_data()`。详情可查阅 `include/linux/bio.h` 和 `Documentation/block/biodoc.txt`。

包含在 `biovec` 和页面结构中的某些字段组合起来有助于确定应当向块设备读写什么数据,而如何读写设备这些更底层的细节则依赖于块设备驱动程序所使用的硬件。

既然已经明了该如何遍历 BIO 结构,那么就必须弄清楚该如何遍历 BIO 结构中的请求结构体链表。这需要用到另外一个宏 `rq_for_each_bio`:

```
-----
include/linux/blkdev.h
495 #define rq_for_each_bio(_bio, rq) \
496     if ((rq->bio)) \
497         for (_bio = (rq->bio); _bio; _bio = bio->bi_next)
-----
```

第 495 行: `bio` 是指当前 BIO 结构, `rq` 是要循环遍历的请求。

每个 BIO 都处理过之后,驱动程序就该更新内核。这一操作是使用函数 `end_that_request_first()` 来完成的。

```
-----
include/linux/blkdev.h
557 extern int end_that_request_first(struct request *, int, int);
-----
```

第 557 行:不出错时,第一个 `int` 参数应当非零,第二个 `int` 参数代表该设备已处理的扇区数。

当 `end_that_request_first()` 函数返回 0 时,所有请求都已处理完毕,可以开始进行清除工作了。这可以调用函数 `blkdev_dequeue_request()` 和 `end_that_request_last()` 来完成,按照这个顺序,这两个函数都把请求当做唯一的参数进行处理。

这时,请求函数已经完成它的工作。接下来,块设备子系统使用块设备驱动程序的请求队列函数来执行磁盘操作。该设备也许还要处理某些 `ioctl` 函数,如处理 RAM 磁盘分区的函数,但这还得取决于块设备的类型。

本节仅略述了块设备的一些基本知识,但 DMA 操作函数、聚集、请求队列命令预操作,以及更先进的块设备的许多特性都没有介绍。要想进一步了解有关信息,可以参阅 `Documentation/block` 目录下的文件。

10.1.8 设备模型和 `sysfs` 文件系统

Linux 2.6 内核的新颖之处在于 Linux 设备模型,这是与 `sysfs` 密切相关的。设备模型存放着与系统中设备及驱动程序相关的一组内部数据。`sysfs` 系统还记录现有的设备,并将其分类:块设备、输入设备、总线设备,等等。系统也记录已有的设备驱动程序以及它们如何与所管理的设备关联。设备模型处于内核内部,而 `sysfs` 就是通向该模型的窗口。由于某些设备和驱动程序

并不通过 sysfs 公开自身，因此最好将 sysfs 理解为内核设备模型的公共视图。

某些设备在 sysfs 中有多项。

尽管设备模型中只有一份数据副本，但正如我们在 sysfs 树中看到的符号链接一样，可以通过多种方式来访问这些数据。

sysfs 文件的层次结构与内核的 kobject 和 kset 结构有关。该模型相当复杂，但绝大多数驱动程序的开发人员不必深究其细节就能完成许多有用的任务^①。当利用 sysfs 属性的概念时，就涉及 kobject，但这是一种抽象的方式。属性是设备或驱动程序模型的组成部分，可以通过 sysfs 文件系统对其访问或修改。属性可能是内部模块变量，用于控制模块对任务的管理，也可能被直接链接从而对各种硬件进行设置。例如，RF 发送器，它工作时有一个基本频率，根据这个基本频率稍作变动就可以实现多个调谐器。我们可以将 RF 驱动程序的模块属性对 sysfs 公开，来改变它的基本频率。

访问某一属性时，sysfs 将调用一个函数来处理该操作。show() 函数用于读取属性，store() 函数用于写入一个属性。show() 和 store() 函数可接收的数据大小最多为一页。

大致了解 sysfs 的工作原理后，可进一步探讨驱动程序通过 sysfs 来注册的细节，公开一些属性，当访问这些属性时，注册要操作的具体函数 show() 和 store()。

我们的首要任务是确定新设备及其驱动程序的类型（例如，usb_device、net_device、pci_device、sys_device 等）。所有这些设备的结构体中都有一个 char*name 字段，sysfs 将使用该字段名在 sysfs 层次结构中标明该设备。

分配完设备结构并为其命名后，就必须创建并初始化 devicer_driver 结构：

```
-----
include/linux/device.h
102 struct device_driver {
103     char      * name;
104     struct bus_type * bus;
105
106     struct semaphore unload_sem;
107     struct kobject kobj;
108     struct list_head devices;
109
110     int (*probe) (struct device * dev);
111     int (*remove) (struct device * dev);
112     void (*shutdown) (struct device * dev);
113     int (*suspend) (struct device * dev, u32 state, u32 level);
114     int (*resume) (struct device * dev, u32 level);
115};
-----
```

第 103 行：name 表示在 sysfs 层次结构中显示的设备驱动程序名。

第 104 行：bus 通常是被自动填充的，驱动程序的编写者不用考虑它。

第 105~115 行：程序员无需设置其他字段的值，它们应当在总线级自动被初始化。

在初始化的过程中，可以调用 driver_register() 来注册设备驱动程序，该函数会把大部

^① 可参阅内核源代码中的文件 documentation/filesystems/sysfs.txt。

分工作交给 `bus_add_driver()` 去做。类似地，设备驱动程序退出时，要调用 `driver_unregister()` 进行注销。

```
-----
drivers/base/driver.c
86 int driver_register(struct device_driver * drv)
87 {
88     INIT_LIST_HEAD(&drv->devices);
89     init_MUTEX_LOCKED(&drv->unload_sem);
90     return bus_add_driver(drv);
91 }
-----
```

注册设备驱动程序之后，可以通过结构 `driver_attribute` 和十分有用的宏 `DRIVER_ATTR` 为驱动程序设置属性：

```
-----
include/linux/device.h
133 #define DRIVER_ATTR(_name, _mode, _show, _store) \
134 struct driver_attribute driver_attr_##_name = { \
135     .attr = {.name = __stringify(_name), .mode = _mode, .owner = THIS_MODULE \
136     }, \
137     .show = _show, \
138     .store = _store, \
139 };
-----
```

第135行：`name` 是设备驱动程序属性的名称，`mode` 是描述属性保护级别的位图。在文件 `include/linux/stat.h` 中包含许多这样的模式，`S_IRUGO`（用于只读操作）和 `S_IWUSR`（用于根用户的写访问）仅是其中的两例。

第136行：`show` 是通过 `sysfs` 读取其属性时所使用的驱动程序函数名。若禁止读取设备的属性，则将 `show` 赋值为 `NULL`。

第137行：`store` 是通过 `sysfs` 写入其属性时所使用的驱动程序的函数名。若禁止写入设备的属性，则将 `store` 赋值为 `NULL`。

为具体的驱动程序实现 `show()` 和 `store()` 的驱动程序函数必须遵循以下函数原型：

```
-----
include/linux/sysfs.h
34 struct sysfs_ops {
35     ssize_t (*show)(struct kobject *, struct attribute *, char *);
36     ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);
37 };
-----
```

回想一下，前面我们说到对 `sysfs` 属性进行读写的限制是 `PAGE_SIZE` 字节。所以驱动程序属性函数 `show()` 和 `store()` 应当确保强制遵守该限制。

现在，可以为内核设备驱动程序添加基本的 `sysfs` 功能了。要想进一步了解有关 `sysfs` 及 `kobject` 的信息，可以查看 `Documentation/device-model` 目录下的文件。

另一种设备驱动程序类型是网络设备驱动程序。网络设备可以发送和接收数据包，它不一定

是一个硬件设备，环回设备就是一个软件网络设备。

10.2 编写代码

10.2.1 设备基础

当你编写了一个设备驱动程序时，该程序通过文件系统中的入口与操作系统进行交互。该入口包含一个主设备号，以便告诉内核访问文件时应当使用哪个驱动程序；同时还有一个从设备号，用于扩大该设备驱动程序的使用范围。加载该设备驱动程序之后，它会注册其主设备号。可以通过查看文件 `/proc/devices` 来获取注册信息：

```
-----
lkp# less /proc/devices
Character devices:
 1 mem
 2 pty
 3 tty
 4 ttyS
 5 cua
 6 lp
 7 vcs
10 misc
29 fb
128 ptm
136 pts
Block devices:
 1 ramdisk
 2 fd
 3 ide0
 7 loop
22 ide1
-----
```

当设备驱动程序向内核注册自己时，其主设备号就会被写入 `/proc/devices` 文件；对于每个字符设备，这一过程都是调用 `register_chrdev()` 函数来实现的。

```
-----
include/linux/fs.h
1: int register_chrdev(unsigned int major, const char *name,
2:      struct file_operations *fops)
-----
```

- **major**: 已注册设备的主设备号。如果 `major` 等于 0，那么内核会动态地为它分配一个与其他已加载的模块不冲突的主设备号。
- **name**: 该字符串表示文件系统 `/dev` 目录下的设备。
- **fops**: 指向 `file-operations` 结构体的指针，它定义了对已注册设备可进行的操作。

如果在创建设备时未设置其主设备号，使用 0 作为主设备号是首选的方法（IDE 驱动程序的主设备号总是为 3，SCSI 为 8，floppy 为 2）。此时需要动态地为它分配一个主设备号，以避免人

为地分配一些可能已使用的主设备号^①。但这样会使得文件系统节点变得稍微复杂一些，因为加载模块后，必须检查该设备的主设备号。例如，检测一个设备时，可能需要进行如下操作：

```
-----
lkp@lkp# insmod my_module.o
lkp@lkp# less /proc/devices
1 mem
...
233 my_module
lkp@lkp# mknod c /dev/my_module0 233 0
lkp@lkp# mknod c /dev/my_module1 233 1
-----
```

这些代码告诉我们如何使用 `insmod` 命令来插入模块。`insmod` 命令可在内核运行时安装可加载的模块。该模块的代码包括：

```
-----
static int my_module_major=0;
...
module_param(my_module_major, int, 0);
...
result = register_chrdev(my_module_major, "my_module", &my_module_fops);
-----
```

前面两行告诉我们如何创建一个值为 0 的默认主设备号，以便需要时进行动态分配，但它允许用户使用变量 `my_module_major` 作为模块的参数，因而不必考虑动态分配。

```
-----
include/linux/moduleparam.h
1: /* This is the fundamental function for registering boot/module
   parameters. perm sets the visibility in driverfs: 000 means it's
   not there, read bits mean it's readable, write bits mean it's
   writable. */
...
/* Helper functions: type is byte, short, ushort, int, uint, long,
   ulong, charp, bool or invbool, or XXX if you define param_get_XXX,
   param_set_XXX and param_check_XXX. */
...
2: #define module_param(name, type, perm)
-----
```

在 Linux 以前的版本中，宏 `module_param` 就是宏 `MODULE_PARM`。Linux 2.6 反对这种做法，它要求必须使用 `module_param`。

- ❑ **name**：该字符串用于访问参数的值。
- ❑ **type**：存放在参数名中的值的类型。
- ❑ **perm**：在 `sysfs` 中可以看到的模块参数名。如果不知道 `sysfs` 是什么，就使用 0，表示不能通过 `sysfs.perm` 访问该参数。

^① `register_chrdev()` 函数返回已分配的主设备号。若要为某设备动态分配主设备号，`register_chrdev()` 函数也许在获取该设备号时大有所为。

回想一下，我们曾经传给 `register_chrdev()` 函数一个指向 `fops` 数据结构的指针。它用于告诉内核驱动程序可以处理什么函数。我们只声明那些模块处理的函数。为了在注册的设备中将 `read`、`write`、`ioctl`、`open` 声明为有效操作，需要添加如下代码：

```
-----
struct file_operations my_mod_fops = {
    .read = my_mod_read,
    .write = my_mod_write,
    .ioctl = my_mod_ioctl,
    .open = my_mod_open,
};
-----
```

10.2.2 符号输出

编写一个复杂的设备驱动程序时，也许要输出驱动程序中定义的某些符号，以便让内核中其他模块使用。这通常用于低级的驱动程序，以便根据这些基础功能构建更高级的驱动程序。

任何设备驱动程序加载后，其所有已输出的符号都会存放内核符号表中，而随后加载的设备驱动程序就可以使用这些由先前的驱动程序输出的符号。当模块之间相互依赖时，加载的顺序就显得至关重要。如果高层模块所需的符号不存在的话，`insmod` 操作就会失败。

在 Linux 2.6 内核中，设备程序员可用如下两个宏来输出符号：

```
-----
include/linux/module.h
187 #define EXPORT_SYMBOL(sym)          \
188     __EXPORT_SYMBOL(sym, "")
189
190 #define EXPORT_SYMBOL_GPL(sym)        \
191     __EXPORT_SYMBOL(sym, "_gpl")
-----
```

使用宏 `EXPORT_SYMBOL` 将符号输出到内核符号表时，允许内核中的任意模块使用这些给定的符号，但使用宏 `EXPORT_SYMBOL_GPL` 时，只有当模块的 `MODULE_LICENSE` 属性中定义了 GPL 兼容许可时，才能够使用这些符号。（该许可的完整列表可以参阅文件 `include/linux/module.h`。）

10.2.3 IOCTL

迄今为止，我们研究的设备驱动程序都是主动操作，或向设备进行数据的读写操作。那么，当一个设备驱动程序不只具有读写功能时，会怎么样呢？若设备驱动程序可以进行不同类型的读写操作时又会怎样呢？又或者当设备需要某种硬件控制接口时会怎样呢？在 Linux 中，设备驱动程序解决这些问题的常用方式就是使用 `ioctl`。

`ioctl` 是一个系统调用，它允许设备驱动程序处理用于控制 I/O 通道的特殊命令。设备驱动程序的 `ioctl` 调用必须遵循 `file_operations` 结构中的如下声明：

```
-----
include/linux/fs.h
863 struct file_operations {
```



```
...
872 int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
-----
```

在用户空间中，ioctl 函数调用定义如下：

```
int ioctl (int d, int request, ...);
```

用户空间定义的第三个参数对于内存来说是一个无类型的指针，这就是把数据从用户空间传递给设备驱动程序的 ioctl 实现的方法。也许看上去很复杂，但在驱动程序中实际应用 ioctl 时相当简单。

首先，为设备声明有效的 IOCTL 号。我们可以参阅文件 Documentation/ioctl-number.txt 并选择一段机器不会使用的代码。查阅当前的 2.6 文件，可以看到 ioctl 代码中字符 'g' 没有被使用。所以，在我们的设备驱动程序中，我们这样声明该字符：

```
#define MYDRIVER_IOC_MAGIC 'g'
```

对于每个设备驱动程序接收到的独特的控制信息，我们都要为它声明一个特有的 ioctl 号。它并不是基于已定义的魔数：

```
-----
#define MYDRIVER_IOC_OP1 _IO(MYDRIVER_IOC_MAGIC, 0)
#define MYDRIVER_IOC_OP2 _IOW(MYDRIVER_IOC_MAGIC, 1)
#define MYDRIVER_IOC_OP3 _IOR(MYDRIVER_IOC_MAGIC, 2)
#define MYDRIVER_IOC_OP4 _IORW(MYDRIVER_IOC_MAGIC, 3)
-----
```

使用 include/asm/ioctl.h 文件中定义的宏 MYDRIVER_IOC_MAGIC 给已列出的四种操作 (op1、op2、op3 以及 op4) 分别分配一个唯一的 ioctl 号。其中宏 MYDRIVER_IOC_MAGIC 是一个 ioctl 魔数。该文档说明了所有信息的含义：

```
-----
Documentation/lioctl-number.txt
6 If you are adding new ioctls to the kernel, you should use the _IO
7 macros defined in <linux/ioctl.h>:
8
9 _IO an ioctl with no parameters
10 _IOW an ioctl with write parameters (copy_from_user)
11 _IOR an ioctl with read parameters (copy_to_user)
12 _IORW an ioctl with both write and read parameters.
13
14 'Write' and 'read' are from the user's point of view, just like the
15 system calls 'write' and 'read'. For example, a SET_FOO ioctl would
16 be _IOW, although the kernel would actually read data from user space;
17 a GET_FOO ioctl would be _IOR, although the kernel would actually write
18 data to user space.
-----
```

从用户空间可以像下面这样调用 ioctl 命令：

```
-----
ioctl(fd, MYDRIVER_IOC_OP1, NULL);
ioctl(fd, MYDRIVER_IOC_OP2, &mydata);
-----
```



```
ioctl(fd, MYDRIVER_IOC_OP3, mydata);
ioctl(fd, MYDRIVER_IOC_OP4, &mystruct);
```

用户空间的程序必需知道 `ioctl` 命令是什么（本例中有 `MYDRIVER_IOC_OP1` 到 `MY_DRIVER_IOC_OP4`）以及这些命令需要什么类型的参数。我们可以通过使用 `ioctl` 系统调用的返回代码来返回一个值，或者将这个参数当作一个需要设置或读入的指针进行解释。对于后一种情况，要注意的一点是，使用指针会涉及将用户空间内存的字节移入/移出内核空间的操作。

使用 `ioctl` 函数来实现内存存在用户空间和内核空间的交换，最简单的方法就是使用 `put_user()` 以及 `get_user()` 函数，这些函数的定义如下：

```
-----
#include/asm-i386/uaccess.h
* get_user: - Get a simple variable from user space.
* @x: Variable to store result.
* @ptr: Source address, in user space.
...
* put_user: - Write a simple value into user space.
* @x: Value to copy to user space.
* @ptr: Destination address, in user space.
-----
```

函数 `put_user()` 和 `get_user()` 确保要读取或写入用户空间的内存，在调用这些函数时便已在内存中。

也许你还想为设备驱动程序的 `ioctl` 函数添加另一个约束条件，那就是身份验证。

检查调用 `ioctl` 函数的进程是否具有调用 `ioctl` 的权限，通常使用权能。驱动程序身份验证中常用的权能是 `CAP_SYS_ADMIN`：

```
-----
include/linux/capability.h
202 /* Allow configuration of the secure attention key */
203 /* Allow administration of the random device */
204 /* Allow examination and configuration of disk quotas */
205 /* Allow configuring the kernel's syslog (printk behavior) */
206 /* Allow setting the domainname */
207 /* Allow setting the hostname */
208 /* Allow calling bdflush() */
209 /* Allow mount() and umount(), setting up new smb connection */
210 /* Allow some autofs root ioctls */
211 /* Allow nfsservctl */
212 /* Allow VM86_REQUEST_IRQ */
213 /* Allow to read/write pci config on alpha */
214 /* Allow irix_prctl on mips (setstacksize) */
215 /* Allow flushing all cache on m68k (sys_cacheflush) */
216 /* Allow removing semaphores */
217 /* Used instead of CAP_CHOWN to "chown" IPC message queues, semaphores
218 and shared memory */
219 /* Allow locking/unlocking of shared memory segment */
220 /* Allow turning swap on/off */
221 /* Allow forged pids on socket credentials passing */
222 /* Allow setting readahead and flushing buffers on block devices */
-----
```

```

223 /* Allow setting geometry in floppy driver */
224 /* Allow turning DMA on/off in xd driver */
225 /* Allow administration of md devices (mostly the above, but some
226 extra ioctls) */
227 /* Allow tuning the ide driver */
228 /* Allow access to the nvram device */
229 /* Allow administration of apm_bios, serial and bttv (TV) device */
230 /* Allow manufacturer commands in isdn CAPI support driver */
231 /* Allow reading non-standardized portions of pci configuration space */
232 /* Allow DDI debug ioctl on sbpcd driver */
233 /* Allow setting up serial ports */
234 /* Allow sending raw qic-117 commands */
235 /* Allow enabling/disabling tagged queuing on SCSI controllers and sending
236 arbitrary SCSI commands */
237 /* Allow setting encryption key on loopback filesystem */
238
239 #define CAP_SYS_ADMIN 21
-----

```

include/linux/capability.h 中定义了许多其他更具体的权能，它们也许更适合权限要求更严的设备驱动程序，但 CAP_SYS_ADMIN 已包罗万象了。

为了在驱动程序中检查调用进程的权能，可以添加类似下面这样的代码：

```

if (! capable(CAP_SYS_ADMIN)) {
    return EPERM;
}

```

10.2.4 轮询与中断

当设备驱动程序向它控制的设备发出命令时，可以通过两种方式来判断该设备是否已经成功接收到命令，分别是设备轮询方式和设备中断方式。

轮询方式意味着设备驱动程序要周期性地检查设备的状态，以确保相应设备成功地接收命令。由于设备驱动程序是内核的一部分，如果直接轮询访问设备，可能会带来灾难性的后果，内核将在此过程中无所事事，直到设备完成轮询操作。设备驱动程序可以通过使用系统定时器来避免这种现象。当设备驱动程序要轮询访问设备时，它将在稍后调度内核，以调用设备驱动中的某个例程来检查设备状态，而无须让内核闲置。

深入探讨内核中断的工作原理之前，必须先来研究锁定访问内核临界区代码的主要方法：自旋锁。自旋锁的作用就是在进入临界区之前为一些特定的标志位设置一定的值，在退出临界区之后再复位这些标志位。当进程上下文不能阻塞在内核代码中就是这种情况，应该使用自旋锁，我们来看看 x86 及 PPC 体系结构中有关自旋锁的代码。

```

-----
include/asm-i386/spinlock.h
32 #define SPIN_LOCK_UNLOCKED (spinlock_t) { 1 SPINLOCK_MAGIC_INIT }
33
34 #define spin_lock_init(x) do { *(x) = SPIN_LOCK_UNLOCKED; } while(0)
...
43 #define spin_is_locked(x) (*(volatile signed char *)&(x)->lock) <= 0)

```

```

44 #define spin_unlock_wait(x)  do { barrier(); } while(spin_is_locked(x))

include/asm-ppc/spinlock.h
25 #define SPIN_LOCK_UNLOCKED  (spinlock_t) { 0 SPINLOCK_DEBUG_INIT }
26
27 #define spin_lock_init(x)  do { *(x) = SPIN_LOCK_UNLOCKED; } while(0)
28 #define spin_is_locked(x)  ((x)->lock != 0)
while(spin_is_locked(x))
29 #define spin_unlock_wait(x)  do { barrier(); } while(spin_is_locked(x))
-----

```

在 x86 平台下, 当未加锁时自旋锁的 flag 值实际为 1, 而在 PPC 平台下, 这时 flag 的值为 0。由此可见, 编写一个驱动程序时, 应当使用宏而不是原始值, 以确保平台间的兼容。

想要获得自旋锁的任务将在紧凑循环中不断检测该特殊标志的值, 直到该标志的值小于 0。所以我们说, 等待自旋锁的任务处于自旋状态。(可参阅这两个代码块中的函数 `spin_unlock_wait()`。)

当内核想要执行一段不受任何中断打扰的临界区代码时, 驱动程序通常会在处理中断时使用自旋锁。在 Linux 以前的版本中, 使用函数 `cli()` 和 `sti()` 来禁用和启用中断。从 2.5.28 版开始, 逐渐用自旋锁代替了这两个函数, 这种新的方式使用如下代码来保证执行临界区代码时不被中断:

```

-----
Documentation/cli-sti-removal.txt
1: spinlock_t driver_lock = SPIN_LOCK_UNLOCKED;
2: struct driver_data;
3:
4: irq_handler (...)
5: {
6: unsigned long flags;
7: ....
   spin_lock_irqsave(&driver_lock, flags);
8: ....
10: driver_data.finish = 1;
11: driver_data.new_work = 0;
12: ....
13: spin_unlock_irqrestore(&driver_lock, flags);
14: ....
15: }
16:
17: ...
18:
19: ioctl_func (...)
20: {
21: ...
22: spin_lock_irq(&driver_lock);
23: ...
24: driver_data.finish = 0;
25: driver_data.new_work = 2;
26: ...
27: spin_unlock_irq(&driver_lock);

```



```
28: ...
29: }
```

第 8 行：进入临界区之前，保存中断标表并对 `driver_lock` 加锁。

第 9~12 行：每次仅允许一个任务执行临界区代码。

第 27 行：临界区末尾。恢复中断状态并对 `driver_lock` 解锁。

使用 `spin_lock_irq_save()`（以及 `spin_lock_irq_restore()`）函数，可以确保在中断处理程序工作之前就禁止中断，并在中断处理程序完成工作后仍保持禁止中断的状态。

当使用 `ioctl_func()` 函数对 `driver_lock` 加锁之后，其他调用 `irq_handler()` 的调用者就会自旋。因此，必须确保 `ioctl_func()` 函数中的临界区代码尽快被执行，以保证上半部中断处理程序 `irq_handler()` 等待的时间尽可能短。

让我们来分析以下如何创建一个中断程序及其上半部中断处理程序（参阅 10.2.5 节中的下半部中断处理程序，它使用了工作队列）：

```
-----
#define mod_num_tries 3
static int irq = 0;
...
int count = 0;
unsigned int irqs = 0;
while ((count < mod_num_tries) && (irq <= 0)) {
    irqs = probe_irq_on();
    /* Cause device to trigger an interrupt.
       Some delay may be required to ensure receipt
       of the interrupt */
    irq = probe_irq_off(irqs);
    /* If irq < 0 multiple interrupts were received.
       If irq == 0 no interrupts were received. */
    count++;
}
if ((count == mod_num_tries) && (irq <= 0)) {
    printk("Couldn't determine interrupt for %s\n",
        MODULE_NAME);
}
-----
```

这些代码将会成为设备驱动程序初始化代码的一部分，在没有中断时会执行失败。既然我们有一个中断，那就可以在内核中注册该中断处理程序及其上半部中断处理程序。

```
-----
retval = request_irq(irq, irq_handler, SA_INTERRUPT,
    DEVICE_NAME, NULL);
if (retval < 0) {
    printk("Request of IRQ %n failed for %s\n",
        irq, MODULE_NAME);
    return retval;
}
-----
```

request_irq()函数的原型为:

```
-----
arch/ i386/kernel/irq.c
590 /**
591 * request_irq - allocate an interrupt line
592 * @irq: Interrupt line to allocate
593 * @handler: Function to be called when the IRQ occurs
594 * @irqflags: Interrupt type flags
595 * @devname: An ascii name for the claiming device
596 * @dev_id: A cookie passed back to the handler function
...
622 int request_irq(unsigned int irq,
623     irqreturn_t (*handler)(int, void *, struct pt_regs *),
624     unsigned long irqflags,
625     const char * devname,
626     void *dev_id)
```

irqflags 参数可以是下列宏的 ord 值。

- SA_SHIRQ: 表明该中断可由多个设备共享;
- SA_INTERRUPT: 运行 handler 时禁止本地中断;
- SA_SAMPLE_RANDOM: 表明该中断是否是熵源。

如果中断没有被共享,那么 dev_id 必须等于 NULL;反之,由于 handler 接收了其值,因此 dev_id 通常用来存放该设备地址的数据结构。

当模块退出时,应该使用 free_irq()函数来释放每一个请求的中断:

```
-----
arch/ i386/kernel/irq.c
669 /**
670 * free_irq - free an interrupt
671 * @irq: Interrupt line to free
672 * @dev_id: Device identity to free
...
682 */
683
684 void free_irq(unsigned int irq, void *dev_id)
```

如果 dev_id 是一个共享的 irq,那么模块就要确保在调用这个函数之前禁用中断。而且,在中断上下文中绝不允许调用 free_irq()函数。在模块的清除例程中调用 free_irq()函数是我们一贯的做法(参阅函数 spin_lock_irq()及 spin_unlock_irq())。

这时,我们已经注册了中断处理程序以及相关的 irq。现在就必须编写上半部中断处理程序了,我们把它定义为 irq_handler():

```
-----
void irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
    /* See above for spin lock code */
    /* Copy interrupt data to work queue data for handling in
```

```

    bottom-half */
    schedule_work( WORK_QUEUE );
    /* Release spin_lock */
}
-----

```

如果你仅仅需要一个快速中断处理程序，可以用 tasklet 来代替工作队列：

```

-----
void irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
    /* See above for spin lock code */
    /* Copy interrupt data to tasklet data */
    tasklet_schedule( TASKLET_QUEUE );
    /* Release spin_lock */
}
-----

```

10.2.5 工作队列和 tasklet

中断处理程序中的大多数任务都是由工作队列完成的。在上一节中，我们看到中断处理程序的上半部会将中断的相关数据从中断复制到一个数据结构中，之后再调用 `schedule_work()` 函数。

为了让任务从工作队列中运行，任务必须封装在 `work_struct` 结构中。为了在编译时就声明任务结构体，可以使用 `DECLARE_WORK()` 宏。例如，我们可以把下列代码添加到模块中，用相关的函数和数据初始化任务结构体。

```

-----
...
struct bh_data_struct {
    int data_one;
    int *data_array;
    char *data_text;
}
...
static bh_data_struct bh_data;
...
static DECLARE_WORK(my_mod_work, my_mod_bh, &bh_data);
...
static void my_mod_bh(void *data)
{
    struct bh_data_struct *bh_data = data;

    /* all the wonderful bottom half code */
}
-----

```

中断处理程序的上半部将通过 `my_mod_bh` 在 `bh_data` 中设置需要的所有数据，然后调用 `schedule_work(my_mod_work)` 函数。

`schedule_work()` 函数可用于任何模块，但这只是说，在通用的工作队列“事件”中我们会用到工作调度。某些模块可能想创建自己的工作队列，但是一个工作队列所需的函数只输出到

GPL 兼容的模块。因此，如果想保留模块的所有权的话，就必须使用通用的工作队列。

可以使用 `create_workqueue()` 宏来创建一个工作队列，该函数实际上是以第二个参数为 0 调用 `__create_workqueue()`。

```
-----
kernel/workqueue.c
304 struct workqueue_struct *__create_workqueue(const char *name,
305                                             int singlethread)
-----
```

`name` 可以多达 10 个字符。

如果 `singlethread` 等于 0，内核将为每个 CPU 创建一个 `workqueue` 线程；反之，若 `singlethread` 等于 1，内核仅为整个系统创建一个 `workqueue` 线程。

创建任务结构体的方式与前面介绍的类似，但将使用 `queue_work()` 函数而不是 `schedule_work()` 函数将它们置于自定义的工作队列中。

```
-----
kernel/workqueue.c
97 int fastcall queue_work(struct workqueue_struct *wq, struct work_struct
*work)
98 {
-----
```

`wq` 是通过 `create_workqueue()` 函数创建的自定义工作队列。

`work` 是添加到 `wq` 队列中的任务结构体。

我们可以在 `kernel/workqueue.c` 中看到如下其他工作队列函数：

- ❑ **queue_work_delayed()**。确保在指定 jiffies 数达到之前不调用该任务结构体函数。
- ❑ **flush_workqueue()**。使调用者等待，直到队列中所有被调度的任务都已完成。这通常在设备驱动程序退出时才使用。
- ❑ **destroy_workqueue()**。刷新并且释放工作队列。

类似的函数 `schedule_work_delayed()` 和 `flush_scheduled_work()` 用于通用的工作队列。

10.2.6 增加系统调用的代码

我们可以通过编辑 `/kernel` 目录下 `makefile` 文件来添加一个包含我们自己函数的文件，但更简单的方法是在源代码树中已经存在的文件内加入我们的函数代码。文件 `/kernel/sys.c` 中包含了用于系统调用的内核函数，文件 `arch/i386/kernel/sys_i386.c` 中以非标准的调用顺序给出了 x86 下的系统调用。我们可以将用 C 编写的系统调用函数的源代码添加到 `/kernel/sys.c` 文件中，这些代码将在内核态运行，并完成所有预定的工作。在这个过程中要做的其他事情就是支持用户使用该函数。通过 x86 的异常处理程序对函数进行分派。

```
-----
kernel/sys.c
1: ...
2: /* somewhere after last function */
-----
```



```

3:
4: /* simple function to demonstrate a syscall. */
5: /* take in a number, print it out, return the number+1 */
6:
7: asmlinkage long sys_ourcall(long num)
8: {
9:     printk("Inside our syscall num =%d \n", num);
10:    return(num+1);
11: }
-----

```

当异常处理程序处理 `int 0x80` 异常时，它就会在系统调用表中进行索引。在文件 `/arch/i386/kernel/entry.S` 中包含了底层的中断处理例程和系统调用表 `sys_call_table`。该系统调用表用汇编代码实现了 C 语言中的数组，表中每个元素都是 4 字节，并且都被初始化为一个函数地址。按照惯例，必须在这些函数的名字前加上前缀 `sys_`。因为该表中函数的位置决定了它的系统调用号，所以必须将我们的函数名添加到该表末尾。下面是添加修改后该表的内容：

```

-----
arch/i386/kernel/entry.S
: .data
608: ENTRY(sys_call_table)
    .long sys_restart_syscall /* 0 - old "setup()" system call, used for
restarting*/
...
    .long sys_tgkill /* 270 */
    .long sys_utimes
    .long sys_fadvise64_64
    .long sys_ni_syscall /* sys_vserver */
    .long sys_ourcall /* our syscall will be 274 */
884: nr_syscalls=(.-sys_call_table)/4
-----

```

文件 `include/asm/unistd.h` 将系统调用和它们在 `sys_call_table` 中的位置编号关联起来。而且该文件中的宏例程有助于用户程序（用 C 语言编写的）把参数装入寄存器，下面是插入到我们的系统调用中的对 `unistd.h` 的修改：

```

-----
include/asm/unistd.h
1: /*
2: * This file contains the system call numbers.
3: */
4:
5: #define __NR_restart_syscall 0
6: #define __NR_exit 1
7: #define __NR_fork 2
8: ...
9: #define __NR_utimes 271
10: #define __NR_fadvise64_64 272
11: #define __NR_vserver 273
12: #define __NR_ourcall 274
13:
14: /* #define NR_syscalls 274 this is the old value before our syscall */

```

```
15: #define NR_syscalls 275
```

最后，应当创建一个用户程序来测试这个新的系统调用，正如本节前面所述，当程序员将 C 代码中的参数加载到 x86 的寄存器中时，可以借助于系统提供的一系列宏。在 `/usr/include/asm/unistd.h` 文件中定义了七个宏：`_syscallx(type,name,...)`，其中 `x` 表示参数的个数。每个宏都会根据 0~5 的某个数字加载对应数量的参数，而 `syscall6(...)` 宏只传递一个指向更多参数的指针。以下示例程序使用一个参数。在这个例子中（在第 5 行），使用 `/unistd.h` 文件中的 `_syscall1(type,name,type1,name1)` 宏，这个宏根据适当的参数，通过 `int 0x80` 入口对一个系统调用进行解析：

```
-----
mytest.c
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include "/usr/include/asm/unistd.h"
4:
5: _syscall(long,ourcall,long, num);
6:
7: main()
8: {
9: printf("our syscall --> num in=5, num out = %d\n", ourcall(5));
10: }
```

10.3 构建和调试

在内核中添加代码后就需要不断运行，修正错误。本节将描述如何调试你编写的内核代码以及如何构建与调试相关的工具。

调试设备驱动程序

在前述章节中，我们使用 `/proc` 文件系统来搜集有关内核的信息。我们同样可以通过 `/proc` 文件系统来获得与设备驱动程序有关的信息，以便于用户访问，而且是调试设备驱动程序的好方法。当对 `/proc` 文件系统进行读写操作时，它的每个节点都连接到一个内核函数。在 Linux 2.6 内核中，对内核部分（包括设备）的大多数写操作都是通过 `sysfs` 而不是 `/proc` 文件系统来完成的。在内核运行的过程中，这些操作修改某个具体内核对象的属性。`/proc` 文件系统在只读操作中仍然非常有用，不过这些只读操作需要处理大量数据而不太关心对象的属性——值对。本节只讨论从 `/proc` 文件系统中读入数据的情况。

要想对你的设备进行读访问，首先就要在 `/proc` 文件系统中创建一个节点^①，这可以通过函数 `create_proc_read_entry()` 来实现：

```
-----
include/linux/proc_fs.h
```

① 每一个节点在整个目录树中，是一个文件或是一个目录。——译者注

```

146 static inline struct proc_dir_entry *create_proc_read_entry(const char
    *name,
147     mode_t mode, struct proc_dir_entry *base,
148     read_proc_t *read_proc, void * data)
-----

```

*name 是在 /proc 文件系统中出现的节点项，如果 mode 等于 0，表示任何进程都可以读取该文件。如果为单个设备驱动程序创建了多个不同的 proc 文件，最好是先使用函数 proc_mkdir() 建立一个 proc 目录，然后把所有 proc 文件都放到该目录下。*base 是 /proc 下放置 proc 文件的目录路径，如果其值为 NULL，表示文件就直接在 /proc 目录下。读取该文件时，可以调用 *read_proc 函数，*data 则是一个被传回给 *read_proc 的指针：

```

-----
include/linux/proc_fs.h
44 typedef int (read_proc_t)(char *page, char **start, off_t off,
45     int count, int *eof, void *data);
-----

```

这是可通过 /proc 文件系统读取的函数原型。*page 是一个指向缓冲区的指针，该函数会向这个缓冲区写入数据，以便进程读取 /proc 文件的内容。该函数必须从 *page 所指缓冲区中偏移为 off 个字节的地方开始写数据，并且写的数据不能超过 count 个字节。由于绝大多数读操作仅返回少量信息，因此很多实现都忽略了 off 和 count。此外，**start 指针在内核中也极少用到，也可以不考虑它。但如果你实现的读函数返回大量数据，那就要使用 off、count 以及 **start 来管理一次只读入少量信息的数据块。当读操作完成后，该函数应当向 *eof 写入 1，而 *data 将作为参数，传递给在 create_proc_read_entry() 函数中定义的读函数。

10.4 小结

本章覆盖了设备驱动程序、模块以及系统调用等内容，并描述了 Linux 使用设备驱动程序的不同方式。

更准确地说，我们探讨了如下几个主题。

- 描述了 Linux 文件系统上的 /dev 树，并论述了如何确定什么样的设备驱动程序控制什么样的设备。
- 解释了设备驱动程序如何利用文件数据结构和文件操作结构来处理文件系统的 I/O 操作。
- 讨论了用户级内存与内核内存的不同之处，以及设备驱动程序如何在两者之间传递数据。
- 分析了 Linux 内核中等待队列的建立过程，并举例说明当设备驱动程序需要等待特定资源时应该如何使用等待队列。
- 探索了等待队列和中断的原理，这也是需要将 CPU 的控制权交给其他进程时，Linux 内核用来明确中断设备驱动程序处理过程的方法。
- 介绍了 Linux 的系统调用，并概述其基本功能。
- 讨论了块设备驱动程序与字符设备驱动程序的不同之处，并介绍了 linux 2.6 中引入的新设备模型，同时对 sysfs 进行了简要说明。

第 10 章的第一部分，我们抽象地介绍了以上主题，并从以上各个方面分析了特殊的设备驱动

程序/dev/random。第二部分通过具体的例子和示例代码,说明如何构造一个实际的设备驱动程序。更准确地说,可以归纳为以下几点。

- 介绍了如何在/dev目录下创建一个与设备驱动程序关联的节点,以及如何创建动态模块。
- 介绍了 Linux 2.6 中从设备驱动程序模块输出符号的新方法。
- 举例说明设备驱动程序如何提供 IOCTL 函数,这些函数允许设备通过文件系统与内核交互。
- 解释了中断和轮询如何产生,以及自旋锁在 x86 及 PPC 结构体系中的不同。
- 介绍了如何向 Linux 内核添加一个简单的系统调用。

第 10 章为我们在 linux 2.6 版本中开发设备驱动程序打下了坚实的基础,实际上,本书前面的章节也介绍过这些想法和概念。

10.5 习题

1. 阅读第 3 章关于编译内核和用户代码部分,并重新编译内核及 mytest.c 文件,然后运行 mytest.c 文件并观察其输出结果。
2. 给 ourcall 添加一个参数。
3. 在 ourcall 中创建一个系统调用。
4. 解释系统调用与设备驱动程序之间的异同。
5. 为什么不能使用 memcpy 在用户空间和内核空间之间复制数据?
6. 上半部中断处理例程和下半部中断处理例程的区别是什么?
7. tasklet 与 work_queue 有何区别?
8. 当一个设备不仅仅能处理简单的读写请求时, Linux 如何与它进行交互?
9. 未锁定的自旋锁在 x86 平台上的数值是多少? 在 PPC 平台上呢?
10. 请用一句话描述块设备驱动程序与字符设备驱动程序的不同之处。



[G e n e r a l I n f o r m a t i o n]

书名= L i n u x 内核编程

作者= (美) 罗德里格斯, (美) 费舍尔, (美) 斯莫斯基著

页码= 4 0 0

I S B N = 4 0 0

S S 号= 1 2 8 1 6 7 9 4

d x N u m b e r = 0 0 0 0 0 8 1 0 2 6 5 2

出版时间= 2 0 1 1 . 0 5

出版社= 该引擎未能查询到

定价: 7 5 . 0 0

试读地址= <http://book.szdnnet.org.cn/bookDetail.jsp?dxNumber=000008102652&d=2FD8B2B744FE27D2F19554E667C0290F&fenlei=18170403060806&sw=%B1%E0%B3%CC>

全文地址= <http://Wqa.5read.com/image/ss2jpg.dll?did=b1&pid=089F02215B594548D94AAD2DCE8268A706F9BD38CA8B0C4EBF2DC62CA7485254B4F94D81BF0EA2EB25F4AC83F0E02DC46E038E7331C55B47CC75D298411D7152433866735C288A553FE010A65A58A16DD2C57B9C7DA76B888FF5C39FC1CA61438C7C75EAA8440DC7A72B5A91A6F276CD4BE9&jid=/>